DEFENCE
SCIENCE & TECHNOLOGY

State-Machine Modelling in the DOVE System

A. Cant, K. Eastaughffe, C. Liu, B. Mahoney, J. McCarthy and M. Ozols

DSTO-RR-0255

DEFENCE
SCIENCE & TECHNOLOGY

State-Machine Modelling in the DOVE System

A. Cant, K. Eastaughffe*[1]*, C. Liu,
B. Mahony, J. McCarthy and M. Ozols

Information Networks Division
Information Sciences Laboratory

DSTO–RR–0255

ABSTRACT

The DOVE tool supports high-level system modelling and design, and formal reasoning about critical properties. DOVE uses state-machine graphs to illustrate designs, thus building on a familiar and effective means of communicating system designs to a wide audience. DOVE employs a propositional temporal logic to express desirable behavioural properties of the design, and presents it in a sequent calculus syntax for ease of manipulation. A verification procedure which can handle temporal properties of DOVE state machines is included through high level tactics in a graphical proof tool interface. The DOVE program is committed to developing proof visualization techniques to complement the power of this proof scheme. This paper presents the theoretical structure underlying the DOVE tool.

APPROVED FOR PUBLIC RELEASE

[1] Current address: Praxis Critical Systems Limited, 20 Manvers Street, Bath BA1 1PX, United Kingdom.

AQ F03-12-2551

Published by

DSTO Information Sciences Laboratory
PO Box 1500
Edinburgh, South Australia, Australia 5111

Telephone: *(08) 8259 5555*
Facsimile: *(08) 8259 6567*

AR No. AR 012–783
February, 2003

State-Machine Modelling in the DOVE System

EXECUTIVE SUMMARY

Defence work involves a large number of critical systems which require the highest levels of engineering assurance. Safety and security concerns are the most obvious. The achievement of such high levels of assurance must be based on rigorous analysis techniques, the most sophisticated of which make use of formal languages with strictly-defined semantics and are referred to as *formal methods.* Indeed, a number of international standards mandate the rigorous analysis of system designs through the application of formal methods techniques. Thus, there is a clear need for tool support which will facilitate such analyses through the various stages of the design process.

The tool for Design Oriented Verification and Evaluation (DOVE), was developed by the Defence Science and Technology Organisation (DSTO) in Australia to meet this challenge. It provides a simple, but powerful, means of applying formal modelling and verification techniques to the design of safety- and security-critical systems, adopting the ubiquitous state-machine as its basic design model. State machines are a familiar and effective means of communicating system designs to a variety of stakeholders, and are easy to represent and to manipulate with a graphical user interface. DOVE also makes strong use of the Isabelle proof tool: from parsing of user input, through a definitional embedding of the entire logical model for state machine analysis, to the statement and verification of critical properties.

The aims of the DOVE project are fourfold:

1. to answer the challenge of providing appropriate tool support for machine design;
2. to provide a temporal logic sufficiently rich to specify critical properties of the machine, and to support a sufficiently powerful proof methodology for their verification;
3. to provide a visual interface between the user and the proof tool, thereby enhancing the trust in the analysis; and,
4. to introduce sufficient tactical structure so that the proof methodology is at a user-friendly level.

This paper details the system-description language, and the temporal logic language used to describe system properties. It describes the logical model in which the state machine can be analysed, and the calculational and tactical framework for verification of the system properties.

Authors

Tony Cant
TCS group

Tony Cant, Ph.D., is a Senior Research Scientist within the Trusted Computer Systems Group of DSTO. He leads a section that carries out research into high assurance methods and tools, focusing on approaches to machine-assisted reasoning about their critical properties. He also provides technical and policy advice to the Defence Materiel Organisation on safety management issues, and is the editor-in-chief of the Def (Aust) 5679 Standard entitled "The Procurement of Computer-Based Safety Critical Systems", published by the Land Engineering Agency.

Katherine A. Eastaughffe
Praxis Critical Systems Limited

Katherine Eastaughffe joined DSTO in 1990, after completing a BSc (Hons) at the University of Queensland. In 1995 she completed a DPhil at University of Oxford, UK, on a topic involving mathematical logic, type theory and functional programming. Katherine researched theorem prover interfaces and security protocol verification at Cambridge University before leaving to apply her skills in an industry setting.

Chuchang Liu
TCS group

Chuchang Liu joined DSTO in 1998 and is a Research Scientist with the Trusted Computer Systems group. His research interests include computer network security, formal methods for software specification and verification, and temporal logics and their applications. Dr. Liu obtained a PhD in computing science from Macquarie University in 1998. His BSc(Hons) and MEcomp degrees were obtained from Wuhan University, China, where he worked as a Lecturer/Associate Professor for nine years.

Brendan Mahony

TCS group

Brendan Mahony has actively investigated the application of formal mathematics to the development of trusted systems in a number of areas, including multi-level secure systems, security protocols, real-time, concurrent, and reactive systems since 1988. He was awarded a PhD in real-time systems verification from the University of Queensland in 1991 and has worked for the Trusted Computer Systems group at the DSTO since 1995.

Jim McCarthy

TCS group

Jim McCarthy came to the Trusted Computer Systems group in 1998, via a career in theoretical physics dating back to his Ph.D. at Rockefeller University in 1985. He is working as a Research Scientist to develop high assurance methods and tools, to model specific (typically infosec) critical systems, and to monitor progress in Quantum Computation.

Maris Ozols

TCS group

Maris Ozols is a Senior Research Scientist within the Trusted Computer Systems group. His research interests include methods and tools for trusted systems design, development and evaluation. Prior to joining DSTO in 1990, Maris completed a BSc (Hons) at the University of Adelaide, followed by mathematical research in algebra at Monash University, obtaining his PhD in 1991.

Contents

Appendices

Figures

Chapter 1

Introduction

In a critical system (e.g., safety-critical or security-critical), if a process misses some deadlines or fails to respond to some important events then a fault may develop. The consequences in a real-world system could include an accident, a security breach, or other corresponding undesirable results. Many faults are introduced at the requirements-capture and design levels, and are propagated through further development. The first line of defence against such disasters is a clear and precise design, with trustable analysis to verify that the design implements desired critical properties. The critical properties themselves must be clearly stated, and – since critical systems are most effectively specified by their ongoing behaviour – this is best done in a temporal logic.

Our previous work [6, 10] has shown that formal methods – such as formal specification, along with design and code verification – can provide the highest level of assurance for critical systems, and should therefore be used throughout the design stage. Indeed, their use is mandated by a number of international standards (for example UK Defence Standards 00-55 [12] and 00-56 [13]; ITSEC [11]; and Def(AUST) Standard 5679 [1]).

The DOVE (Design Oriented Verification and Evaluation) system is a powerful design tool which is being developed to support such analysis. The over-riding principle in DOVE's construction, originally proposed in [2], is to provide a minimalist approach to the application of formal methods. In other words, it should aid the different participants of the design process without significant disruption of their standard practice. At the same time, the formal analysis must be consistent at all levels of the tool – from the tool itself, to the theories it produces for the user to reason with.

One manifestation of these principles is the use of state-machine graphs to present designs. The state machine paradigm is a widely-used system design model, forming an important component of many existing approaches. State machines are particularly suitable in interactive tool support for the design process. Not only can the diagrammatic representation of a state machine effectively communicate the design to the many different participants in the design process, they are also a powerful and flexible model of computation in which a wide range of design issues can be treated at a convenient level of abstraction. In particular, they are amenable to the application of powerful automated tools such as silicon compilers, as well as formal analysis techniques. A high level of assurance can be gained by a combination of two aspects: detailed evidence of a machine-checked formal proof, together with a high-level argument at the diagrammatic level that can be easily understood by, and communicated to, humans.

DOVE state machines are much more general than what is known in the literature as a "finite state machine". Indeed, the allowed variable types are essentially arbitrary, so there is no restriction to a finite configuration space.

Another application of the development principles, which is somewhat hidden from the user, is the fact that the logical structure for the DOVE tool is definitionally embedded in the Higher-Order Logic (HOL) of the Isabelle [34] proof tool. Moreover, the theories the tool produces for reasoning about a given state machine are built directly on top of this structure. From the parsing of user input to the application of tactics in verification, all analysis is directly carried out in the Isabelle proof tool.

Adopting the state machine paradigm, the aims of the DOVE project are fourfold:

1. to answer the challenge of providing appropriate tool support for machine design;
2. to provide a temporal logic sufficiently rich to specify critical properties of the machine, and to support a sufficiently powerful proof methodology for their verification;
3. to provide a visual interface between the user and the proof tool, thereby enhancing the trust in the analysis; and,
4. to introduce sufficient tactical structure so that the proof methodology is at a user-friendly level.

We now outline where these aims are elaborated in the present paper.

The tool support architecture is outlined in Chapter 2. Later sections contain the concrete syntax which enables user interaction with the DOVE tool. In particular, the system-description language used for specifying state machines appears in Chapter 3, along with an example of a theory for traffic lights design.

Chapter 4 defines the language for describing machine properties. This language is a temporal logic, its semantics is a trace-based extension of that of the system-description language. A temporal property is then a predicate over histories. The temporal operators are defined in this model, and a sequent calculus structure is established. Reasoning about machine properties can then be pursued in the temporal logic model, once it has been restricted from arbitrary histories (traces) to actual machine executions as informed by the state machine.

Chapter 5 collects the calculational constructs required for verifying machine properties. We establish a sequent calculus for representing and manipulating the properties. The sequent calculus is very good for "static" properties; i.e., for dealing with the predicate logic structure. However, we propose that the temporal aspects be handled by an inductive application of high level symbolic animation techniques such as back-substitution. Thus we introduce the notion of predicate transformers, and a scheme which iteratively constructs the weakest invariant which will imply a desired property (or will negate a false property in a finite number of iterations).

In the current version of DOVE a variant of this scheme is implemented, as explained in Chapter 6. As an example, certain properties of the traffic lights design are verified.

Finally, Chapter 7 contains a discussion of the related literature, and concludes with remarks on future development of the tool.

The manner in which the third and fourth aims are implemented by these constructions is briefly mentioned at various points throughout the paper. We leave further elaboration to a later publication. However, it is worthwhile commenting here on the nature of formal analysis with interactive proof assistants. By the nature of highly-interactive computer proof assistants, each proof step is typically "small" – making it difficult to grasp a higher level view of what a given sequence of steps achieves. On the other hand, while for automatic theorem provers the required user-input is typically minimal, they generally provide little visibility of the proof. This leads to the serious drawback that, when they fail to prove a result, essentially no information is given to the user. A user then needs to assess how the required result could be obtained by partitioning it with lemmas – this requires a deep understanding of the theorem prover.

Proof steps which are too small can frustrate the user, while those that are too big can disorient the user by their severe alterations of the proof state. It is important that a computer proof assistant can not only perform trivial tasks unguided – thus displaying some levels of automation – but should also allow proof steps controlled by the user that are predictable, comprehensible, and natural in the given domain. There is a lack of tools that moderate between these highly interactive and highly automated extremes. Thus, it is the aim of the DOVE project to build tactics which provide proof commands at a medium level of granularity. The basic structure of these tactics will be evident from the discussions in this paper.

The approach we employ in implementing the DOVE tool is rather generic, and it may therefore be of interest for the design and analysis of other such tools. Moreover, the solid treatment given here of the corresponding theoretical underpinnings, and also of the proposed methodologies, may aid the user to apply the DOVE tool more effectively in specific applications.

Chapter 2

Architecture of the DOVE system

The architecture of the DOVE tool is reviewed in this section. This is a brief overview only, as desired background for a proper understanding of the paper. Further details about the tool can be found in the user manual [2].

As shown in Figure 2.1, the DOVE system has three main components:

- The *state-machine editor*, a graph editor used for constructing state-machine diagrams;
- The *state-machine animator* that is used for exploring symbolic execution of a state machine; and
- The *prover* that is used for verifying critical properties of a given machine.

It has been built using three existing tools and/or languages: the Tcl/Tk script language [31], the functional programming language ML [33], and the proof-checking system Isabelle [34]. The user interface providing the graphical representation of the state machine is implemented in Tcl and Tk. The associated process-communication language, Expect [20], is used to implement communication with the semantic representation of the machine in ML, and with proofs carried out in Isabelle. Proof-system level visualisation, and pretty printing of formulae, are provided by the XIsabelle interface, while application-domain level visualisation of proof using the state-machine diagram is implemented in Tcl/Tk. The high-level tool use is straightforward:

- Using the state-machine editor, a state-machine design is set out. A state-machine diagram consists of labelled nodes representing the allowed states the machine can be in at any given time instant, and labelled directed edges between the nodes representing the allowed transitions.
- The state machine definition continues by entering desired types, variables and constants, and the transition definitions, through the graphical interface.
- Once a state-machine definition has been saved such that all its transitions have been defined, an Isabelle theory for that machine can be automatically generated.

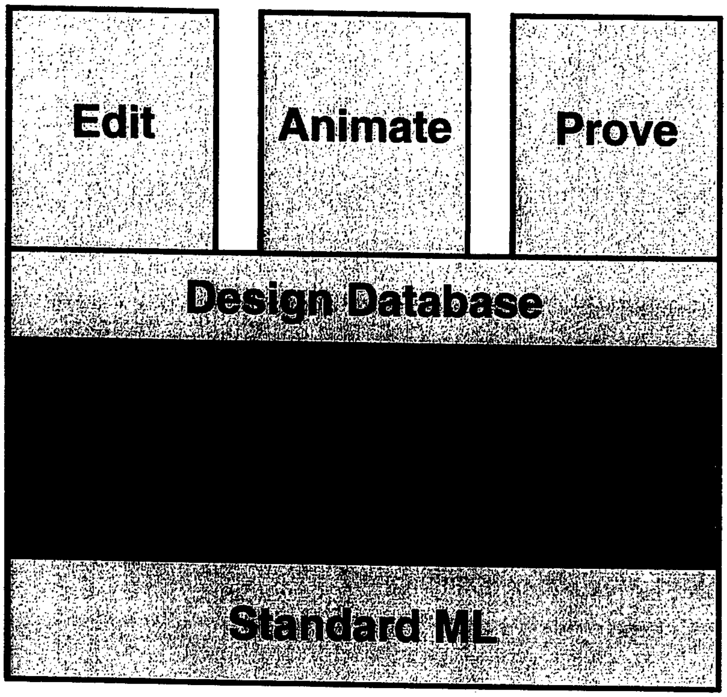

Figure 2.1: The structure of the DOVE tool

- A proof can be commenced by starting up XIsabelle and loading the machine-specific theory in Isabelle, on top of the general DOVE theory for reasoning about state machines and properties.

2.1 Embedding in a theorem prover

The DOVE system uses Isabelle [34], a powerful automated proof tool, to provide the basic logical and bookkeeping features for reasoning about state machine designs. In particular, all parsing of user input is carried out directly in Isabelle, and even an animation step is (automatically) computed as a theorem in Isabelle. Isabelle is a generic logical framework in which a variety of object logics and domain theories can be represented. One of the appeals of Isabelle is that it is quite an "open" system. From the user's point of view this means that essentially arbitrary types and mathematical/physical theories can be supported, thus providing a very powerful tool in principle. From the developer's viewpoint the bonus is that there is much that can be "added on" to improve proof management and structure, thus providing a very useable tool in principle.

The XIsabelle graphical user-interface [32] to Isabelle has been tailored to provide the interactive proof component to DOVE. Proof support includes: a *browser* that displays theory information in a form which is easy to read; a *tactic tree viewer* which graphically represents the overall structure of a proof in progress, and allows easy re-use of proof steps; and a *proof history* which can be displayed, saved, and re-used.

Proof in Isabelle typically works by applying rules in a goal-directed fashion by means of a *subgoal package.* For example, the proposition to be proved can be resolved with the conclusion of a rule to produce subgoals. The main building blocks of goal-directed proof in Isabelle are *tactics*, which apply to a proof state, returning a new proof state. Tactics are composed using *tacticals*, such as THEN, ORELSE and REPEAT. Proof tactics have been formed as a formal proof component of the DOVE system. While the detailed discussion about proof tactics will appear in another paper, a few observations are in order.

- There are quite a number of higher-level logical operations which the Isabelle system and meta-logic provides "for free", and which proved to be very useful. These aspects include unification, beta-reduction in the meta-logic, and general frameworks for rewriting and classical reasoning.
- The relatively low-level basic tactics, such as resolution, form the ideal components to construct the tactics we required. At times, an even lower level of proof steps would be desirable. For example, there are no convenient facilities for focusing on parts of a subgoal, say to do a single rewrite of a subterm. The use of pattern-matching and the ML programming language enables this to be overcome to some degree.
- The generation of a machine specific theory including a set of tactics can be thought of as a *compilation* process. Domain-oriented proof steps are "compiled" into tactics for use in a general-purpose low-level proof checker (Isabelle). More traditional approaches to tactic construction are like *interpreters*, where the tactic is the same for all state machines, and is only dynamically applied to a given specific example. The DOVE approach can have advantages similar to those that program compilation has over interpretation; for example, tactics can be optimised with respect to a specific state-machine.

An effort has been made to use the state-machine diagrams to direct and control proofs in the formal proof procedure. By the use of graphical interfaces at the level of state-machine diagrams, and in the proof tool, DOVE provides some functionality of "proof visualisation" [10]. This can be of great benefit for the user working to understand and communicate the proof, as well as to control its construction.

Chapter 3

A system-description language for state machines

A state machine is modelled to consist of a finite number of states, together with a memory which is a binding of values to variables, and a set of transitions which describe how the state machine may evolve from state to state. This is a common style of description in traditional design methodologies. It also provides an easy way to display the design graphically, by transition graphs in which states are denoted by nodes, and transitions are denoted by directed edges between nodes. We refer to the corresponding diagram as the *state-machine diagram.*

In this section we discuss the system-description language for expressing state-machine designs, beginning with an example of its use in a model of traffic lights – the required syntax is presented in Appendix A. We then explain the abstract model DOVE produces from the user's input.

3.1 Example: the design of the TrafficLights machine

We first present an example of machine specification to illustrate the use of the system-specification syntax. Our belief is that the syntax is clear enough to be understood directly, but we also give a full description in Appendix A.

We call this example the TrafficLights theory. This design models the behaviour of traffic lights controlling a four way intersection. The basic behaviour the system being modelled is easily understood from the state-machine diagram in Figure 3.1: the EW and the NS lights pass alternately through the green-amber-red cycle.

Variables and constants are introduced to describe the obvious traffic-light attributes, and some simple effects of the traffic environment. In particular, the number of cars waiting at a red light is computed (e.g., NCars + Scars for the EW cycle) so that when it exceeds some specified maximum the lights in the other direction will turn amber. Further, this amber phase has a predetermined duration, AmberTimeOut, so that the light changes to red after the time has increased by this amount. Finally, there are alternative "Wait"

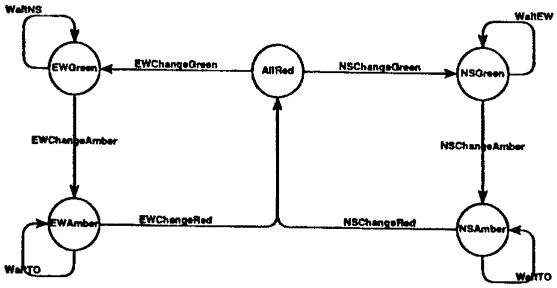

Figure 3.1: A state-machine for traffic lights design

transitions at each of these light changes which are enabled precisely when the actions described above are not. These are included to remove the possibility of deadlock from the system.

The following is the Isabelle/Isar theory file produced automatically[1] by DOVE from the user input.

```
theory TrafficLights = DOVE:

startmachine

states AllRed | EWAmber | EWGreen | NSAmber | NSGreen

datatype Colour = "Red" | "Amber" | "Green"

datatype Direction = "EW" | "NS"

consts
  "MaxCars" :: "nat"
  "MaxTime" :: "nat"

inputs
```

[1] Note that the double quotation marks are required Isabelle syntax.

```
  "ECars" :: "nat"
  "NCars" :: "nat"
  "SCars" :: "nat"
  "WCars" :: "nat"
  "time" :: "nat"

heaps
  "AmberTimeOut" :: "nat"
  "ELight" :: "Colour"
  "LastGreen" :: "Direction"
  "NLight" :: "Colour"
  "SLight" :: "Colour"
  "WLight" :: "Colour"

initpred
  "(NLight = Red) & (SLight = Red) & (ELight = Red) & (WLight = Red)"

transdef "NSChangeGreen" "
  Guard: LastGreen = EW
  Act: NLight <-- Green;
       SLight <-- Green;"

transdef "EWChangeRed" "
  Guard: AmberTimeOut < time
  Act: ELight <-- Red;
       WLight <-- Red;
       LastGreen <-- EW;"

transdef "EWChangeAmber" "
  Let: NSnum <-- (NCars + SCars);
  Guard: MaxCars < NSnum
  Act: ELight <-- Amber;
       WLight <-- Amber;
       AmberTimeOut <-- (time + MaxTime);"

transdef "NSChangeAmber" "
  Let: EWnum <-- (ECars + WCars);
  Guard: MaxCars < EWnum
  Act: NLight <-- Amber;
       SLight <-- Amber;
       AmberTimeOut <-- (time + MaxTime);"

transdef "NSChangeRed" "
  Guard: AmberTimeOut < time
  Act: NLight <-- Red;
       SLight <-- Red;
       LastGreen <-- NS;"
```

```
transdef "EWChangeGreen" "
  Guard: LastGreen = NS
  Act: ELight <-- Green;
       WLight <-- Green;"

transdef "WaitEW" "
  Let: EWnum <-- (ECars + WCars) ;
  Guard: Not (MaxCars < EWnum)
  Act: Skip;"

transdef "WaitNS" "
  Let: NSnum <-- (NCars + SCars) ;
  Guard: Not (MaxCars < NSnum)
  Act: Skip;"

transdef "WaitTO" "
  Guard: Not (AmberTimeOut < time)
  Act: Skip;"


graph "AllRed" "
   EWGreen --EWChangeAmber--> EWAmber
 | AllRed --EWChangeGreen--> EWGreen
 | EWAmber --EWChangeRed--> AllRed
 | NSGreen --NSChangeAmber--> NSAmber
 | AllRed --NSChangeGreen--> NSGreen
 | NSAmber --NSChangeRed--> AllRed
 | NSGreen --WaitEW--> NSGreen
 | EWGreen --WaitNS--> EWGreen
 | EWAmber --WaitTO--> EWAmber
 | NSAmber --WaitTO--> NSAmber"

endmachine

end
```

3.2 Abstract model for state machine design

The DOVE state machine language provides a convenient method of describing complex state machine designs in human readable format. However, it is not the most convenient format for describing or supporting the verification of temporal properties of state machines. For this we introduce an abstract model of state machines.

The observable attributes of an abstract state machine are the current logical state, the last transition, the current values of the input variables, and the current values of the heap variables. Collectively we refer to these observables as the *configuration* of the state machine. The dynamics of a state machine's behaviour are described by three components.

1. An initialisation predicate, determined by the **initpred** command, restricts the allowed initial values of the heap variables.
2. A state-transition graph describes the initial state and the allowed transitions between states.
3. A transition table describes the way in which each transition updates the heap variables. It is determined by the various **transdef** commands, which have three parts:

    - the let declaration which introduces new local variables to be used to abbreviate expressions;
    - the guard which restricts the allowed current values of the input and heap variables for the machine to pass through the transition; and,
    - the action list, the aforementioned parallel assignment to heap variables.

    An empty Guard part is interpreted as True, while an empty Act part is interpreted as the action Skip which assigns all heap variables their value in the current configuration.

Clearly input and heap variables are treated very differently. This is because input variables represent the points at which external entities may influence the machine evolution. Their values may be accessed, but not updated by the machine. In contrast, heap variables hold values that persist from state to state unless explicitly updated via a machine transition. They can be internal to the machine, or can represent points at which the machine communicates to external entities. Their values may be both accessed and updated by the machine.

The other state machine commands serve to determine the types of the various components of the configuration. They also serve to build up a name space for referring to the configuration components in the definition of the state machine dynamics. The way in which a state machine model is built from a state machine definition is described below.

3.2.1 Configuration types

The **states** command causes an Isabelle type stateDT to be declared. The type contains exactly one element for each state of the state machine. Similarly the **transdef** commands collectively cause the declaration of an Isabelle type transitionDT which contains exactly one element corresponding to each **transdef** command.

The types of the various input variables introduced by the **inputs** command are used collectively to declare an Isabelle type inputTY. A value of type inputTY is a composite of the values of the individual inputs. Similarly the types of the various heap variables are used to define a collective type heapTY.

The overall configuration type is then defined to be the cross product of the four component types.

$$\mathsf{configTY} = \mathsf{stateDT} \times \mathsf{transitionDT} \times \mathsf{inputTY} \times \mathsf{heapTY}$$

3.2.2 Namespace environment (R- and L-values)

The various names introduced by the **states**, **transdef**, **inputs**, and **heaps** commands are collected into a database to aid in the interpretation of the state-machine definition.

The correspondence between the state names introduced by the **states** command and the elements of the stateDT type is recorded. A similar correspondence is recorded between the transition names and the elements of transitionDT.

The treatment of the input and heap namespaces is more complex.

- For each input name we record the way in which the value of that particular input may be extracted from a configuration. This function is called the *R-value* of the input, and for input variable I is described by an Isabelle term of type

  $$\mathtt{RI} : \mathsf{configTY} \to \mathcal{X}_{\mathtt{I}},$$

  where $\mathcal{X}_{\mathtt{I}}$ is the value type of input variable I.
- Similarly, we record the way in which the value of each input may be updated in the collective input type. This function is called the *L-value* of the input and for input variable i is represented by an Isabelle term of type

  $$\mathtt{LI} : \mathcal{X}_{\mathtt{I}} \to \mathsf{configTY} \to \mathsf{configTY}.$$
- We also construct the R- and L-values for all heap variables.

The namespace database thus constructed is used to inform the interpretation of the state machine definition proper, by determining the underlying Isabelle terms referred to by each of the names of the states, transitions, inputs, and heaps. In the syntax simply the variable name is used, and whether this requires R- or L-value is obvious from the context. The details of how this interpretation is performed for each of the state machine definition commands is described below.

3.2.3 The abstract state-machine type

Corresponding to the three components which describe its dynamical behaviour, we represent an abstract state-machine as a record with three fields.

1. The field inip :: configTY → $\mathbb{B}$ records the initialisation condition.
2. The field graph :: stateDT × (edge stateDT transitionDT stateDT) list records the state machine control graph. Here edge denotes a ternary type constructor for an abstract "edge" type, while list is the standard unary list-type constructor.

3. The field tranf :: transitionDT → stateDT → stateDT → inputTY → ((configTY → $\mathbb{B}$) × (configTY → configTY)) records the transition table.

We now elaborate in turn the construction of the three fields for a record $\mathcal{M}$ corresponding to a given state machine definition. We will refer to "the state machine $\mathcal{M}$".

3.2.4 The inip field

The boolean expression which the user inputs in the **initpred** command is translated to a predicate on configurations by interpreting each occurrence of an input or heap variable as the corresponding R-value. We will refer to this as the *R-value interpretation* of the expression. Then inip $\mathcal{M}$ is set equal to the resulting predicate.

3.2.5 The graph field

The **graph** command is directly translated into a tuple consisting of the chosen initial state, and a list of edges in the state machine graph.

An edge is modelled as an abstract datatype with a single ternary type constructor edge. For b, e : stateDT and t : transitionDT, we will write the edge, edge $b\ t\ e$, as

$$b \xrightarrow{t} e\,,$$

and say that b is the *begin* state, e the *end* state, and t the *connecting* transition of the edge.

For ease of notation we will introduce a name for the initial state (the machine name to be understood in context),

$$\texttt{InitialState} = \mathsf{fst}(\mathsf{graph}\ \mathcal{M})\,.$$

3.2.6 The tranf field

In the abstract model, state machine transitions are represented as a guard/action pair. The construction takes a number of steps. Consider first the general **transdef** command

```
transdef  t
Let:      X1 <-- x1; X2 <-- x2; ... ;
Guard:    expr
Act:      H1 <-- h1; H2 <-- h2; ... ;
```

The most direct translation gives the guard/action pair with components

$$\mathsf{guard}\ \texttt{t} : \mathsf{configTY} \to \mathbb{B}$$

and

$$\mathsf{action}\ \texttt{t} : \mathsf{configTY} \to \mathsf{configTY}$$

as follows.

- The component guard $\mathtt{t}$ is directly obtained by R-value interpretation of the boolean expression in the Guard part.

- The action corresponding to a term `H<--h` in the Act part is assignment to the heap variable `H`. Specifically, the heap variable `H` on the left of the assignment is interpreted by its L-value, and it updates the current configuration with the value of the expression `h` (obtained by R-value interpretation) evaluated in the current configuration.

- For all required assignments, the corresponding expression must be evaluated in the current configuration. The total updating action is then obtained by iteration,

$$\mathsf{action}\ \mathtt{t} = (\lambda c \bullet \mathsf{LH}_1\ (\mathtt{h}_1\ c)\,(\mathsf{LH}_2\ (\mathtt{h}_2\ c)\ldots c))\,.$$

- Finally, the local variables are replaced with their corresponding expressions (obtained by R-value interpretation) evaluated in the current configuration.

The result is indeed a guard/action pair, in which a given transition action is only determining the values of *heap* variables in the next configuration. However, at any point in the abstract machine evolution the configuration must reflect the current state and the last transition. Thus it is convenient to lift the guard/action pair to fix the correct current state and update the next configuration appropriately.

The input variables, however, are *not* assigned to in a transition definition for the very good reason that the machine has no control over the environment. However, we will see that verification support is streamlined if we take a "closed system" view of the dynamics, in which the input variables are assigned "fresh undetermined values" in each transition. Thus, we must also lift the action to fix a value for all input variables. We will see in the semantics how the fresh input value is obtained.

For each transition name t, the guard/action pair of the state machine $\mathcal{M}$ for the corresponding transition from state b to state e – while updating the input value to i – is then given by

$$\begin{aligned}\mathsf{tranf}\ \mathcal{M}\ t\ b\ e\ i = \ & ((\lambda c \bullet \mathsf{fst}(c) = b\ \&\ \mathsf{guard}\ t\ c),\\ & (\lambda c \bullet (e, t, i, \mathsf{snd}(\mathsf{snd}(\mathsf{snd}(\mathsf{action}\ t\ c))))))\,.\end{aligned}$$

The clumsy use of fst and snd is simply to pick the correct part out of the (resulting) configuration. For later ease of notation we introduce names to denote the different parts of the tranf tuple (with the machine name $\mathcal{M}$ to be understood from the context)

$$\mathcal{G}_{b\underline{t}e} \ = \mathsf{fst}(\mathsf{tranf}\ \mathcal{M}\ t\ b\ e\ i)$$

$$\mathcal{F}^{i}_{b\underline{t}e} \ = \mathsf{snd}(\mathsf{tranf}\ \mathcal{M}\ t\ b\ e\ i)\,.$$

3.3 Relational semantics of the model

An effective interpretation of the abstract state-machine model is that provided by transition systems. A *transition system* over the space of configurations corresponding to a given state-machine $\mathcal{M}$, is specified by an initial configuration predicate $\iota_{\mathcal{M}} : \mathsf{configTY} \rightarrow \mathbb{B}$, together with a *transition relation* $\mathcal{R}_{\mathcal{M}} : \mathsf{configTY} \rightarrow \mathsf{configTY} \rightarrow \mathbb{B}$ between configurations. An *execution* of the state machine $\mathcal{M}$ is a list of configurations such that:

- the initial configuration satisfies $\iota_{\mathcal{M}}$; and,
- any two subsequent configurations in the execution are related by $\mathcal{R}_{\mathcal{M}}$.

We defer the development of the dynamic model until we have introduced the temporal logic. Let us just see here how the interpretation applies in DOVE state-machines.

Relational semantics are appropriate since they naturally allow for nondeterminism. In DOVE state-machines the root of the nondeterminism is through the input variables – we have observed above that the transition relation must not determine the value of the input variables in the next configuration since these describe the environment external to the machine. The transition action in the abstract model is otherwise deterministic. The initial condition[2] for the transition system corresponding to state machine $\mathcal{M}$ is

$$\iota_{\mathcal{M}}\ (s,t,i,m) = (s = \texttt{InitialState}\ \&\ \mathsf{inip}\ \mathcal{M}\ (s,t,i,m))\,.$$

To every edge $\epsilon = {\scriptstyle b \,\underline{t}\, e}$ in $\mathsf{snd}(\mathsf{graph}\ \mathcal{M})$ we can associate a relation ρ_ϵ between configurations,

$$\rho_\epsilon\ c\ c' = \exists\ i \bullet \mathcal{G}_\epsilon\ c\ \&\ c' = \mathcal{F}^i_\epsilon\ c\,.$$

Note that encoding the transition in a guard/action pair with fixed inputs allows us to decompose the transition relation into a guarded assignment. Although the total assignment function deterministic, the input which is assigned, i, is existentially quantified and thus its value left undetermined as required. The total transition relation of the transition system, $\mathcal{R}_{\mathcal{M}}$, is simply the relational disjunction over edges of the individual edge transition relations.

[2] Note that the value of the transition component in the initial configuration is arbitary, except that it must be a transition of the state machine topology. In this way, without requiring a variant form for the initial configuration we cope with the fact that no transition is traversed to produce it.

Chapter 4

A Temporal Language for Properties of State Machines

The properties we wish to establish for critical devices assert that whatever state is reached, no bad thing has happened so far (that is, a fault which may lead to an accident or security breach). In other words, for any finite length of time, a property holds for the duration of that time. Such a property is called a *safety* property. Heuristically, it asserts the absence of undesirable states, in contrast to what are generally called *liveness* properties, which assert that good things do happen [3].

Because we are only interested in safety properties we interpret the behaviour of the machine by its *histories*: *finite* sequences of configurations. Moreover, actual machine *executions* are histories which begin in the initial state satisfying the initial predicate, and each subsequent configuration is then reached by some transition of the given machine. Without loss of generality, then, we only consider non-empty histories. The safety properties of interest will be *execution properties*; i.e., properties whose validity can be tested over a given execution.

In this section we first introduce an abstract syntax model for the language of system properties. Compared to the concrete syntax – presented for convenience in Appendix B – this make the types explicit for ease of semantic translation. The interpretation in the history-based model follows, after which we formalize the notion of property verification.

4.1 Abstract model of the property specification syntax

The syntax for specifying system properties in DOVE can be considered abstractly as introducing the type of temporal logic formulae, constructed as indicated in the following "datatype".

$$\begin{array}{lll}
\texttt{Temp} & ::= & \textbf{true} \mid \textbf{false} \mid \textbf{first} \\
& \mid & \textbf{pred}\langle\!\langle \mathsf{configTY} \to \mathbb{B}\rangle\!\rangle \mid \textbf{Not}\langle\!\langle \texttt{Temp}\rangle\!\rangle \\
& \mid & \textbf{init}\langle\!\langle \texttt{Temp}\rangle\!\rangle \mid \textbf{Previously}\langle\!\langle \texttt{Temp}\rangle\!\rangle \mid \textbf{Always}\langle\!\langle \texttt{Temp}\rangle\!\rangle \\
& \mid & \textbf{Sometime}\langle\!\langle \texttt{Temp}\rangle\!\rangle \mid \textbf{PreviouslyS}\langle\!\langle \texttt{Temp}\rangle\!\rangle \\
& \mid & \langle\!\langle \texttt{Temp}\rangle\!\rangle \wedge \langle\!\langle \texttt{Temp}\rangle\!\rangle \mid \langle\!\langle \texttt{Temp}\rangle\!\rangle \vee \langle\!\langle \texttt{Temp}\rangle\!\rangle \\
& \mid & \langle\!\langle \texttt{Temp}\rangle\!\rangle \Rightarrow \langle\!\langle \texttt{Temp}\rangle\!\rangle \mid \langle\!\langle \texttt{Temp}\rangle\!\rangle \Leftrightarrow \langle\!\langle \texttt{Temp}\rangle\!\rangle \\
& \mid & \langle\!\langle \texttt{Temp}\rangle\!\rangle \textbf{FromThenOn}\langle\!\langle \texttt{Temp}\rangle\!\rangle \mid \langle\!\langle \texttt{Temp}\rangle\!\rangle \textbf{FromThenOnS}\langle\!\langle \texttt{Temp}\rangle\!\rangle \\
& \mid & \textbf{MostRecently}\langle\!\langle \texttt{Temp}\rangle\!\rangle\langle\!\langle \texttt{Temp}\rangle\!\rangle \mid \textbf{MostRecentlyS}\langle\!\langle \texttt{Temp}\rangle\!\rangle\langle\!\langle \texttt{Temp}\rangle\!\rangle \\
& \mid & \forall\langle\!\langle \beta \to \texttt{Temp}\rangle\!\rangle \mid \exists\langle\!\langle \beta \to \texttt{Temp}\rangle\!\rangle \\
& \mid & \textbf{At}\langle\!\langle \mathsf{stateDT}\rangle\!\rangle \mid \textbf{By}\langle\!\langle \mathsf{transitionDT}\rangle\!\rangle
\end{array}$$

We have chosen the notation such that the correspondence between the concrete and abstract syntax is obvious, except possibly in the three cases we now address.

- A nontemporal formula is interpreted as a configuration predicate (by R-value interpretation). A further constructor must be added to lift nontemporal formulae to temporal formulae, and thus we introduce **pred** : $(\mathsf{configTY} \to \mathbb{B}) \to \texttt{Temp}$. As an example, note that the nontemporal formula

  $$\mathtt{x} = 3 \Rightarrow \mathtt{y} < \mathtt{z}\,.$$

  translates to

  $$\mathbf{pred}(\lambda c \bullet R\mathrm{x}\ c = 3) \Rightarrow \mathbf{pred}(\lambda c \bullet R\mathrm{y}\ c < R\mathrm{z}\ c)\,,$$

  where $\mathtt{x}, \mathtt{y}, \mathtt{z}$ are variables of concrete type $\mathbb{N}$.

- Standard binder notation is used for the quantifiers. Thus, we write $(\forall x \bullet P\ x)$ for $(\forall\ P)$ and $(\exists x \bullet P\ x)$ for $(\exists\ P)$.

- The special constructors **At** : $\mathsf{stateDT} \to \texttt{Temp}$ and **By** : $\mathsf{transitionDT} \to \texttt{Temp}$ allow the user limited access to the "logical" part of the configuration, without the problem of needing to be careful in writing sensible expressions. Their types have been made explicit in passing from abstract to concrete syntax.

The proof system will be represented by terms in the sequent calculus style. A *sequent* consists of a list of temporal formulae $\Gamma = \{p_i\}_{i=1}^{n}$, known as the *hypotheses*, and a target formula q, known as the *goal*. Sequents are expressed by terms of the form $p_1, \ldots, p_n \vdash q$, or simply $\Gamma \vdash q$ for brevity. Thus, in this mixfix syntax, $\vdash : \mathsf{Templist} \to \mathbb{B}$.

4.2 Trace-based semantics of the property-specification syntax

The semantics of temporal formulae is determined by interpreting them on *histories*. A history is represented by a (non-empty, finite) list of configurations (i.e., an element of the set $\mathsf{configTY}^{+}$). A temporal formula is interpreted as a history predicate – a function from histories to truth values. We will use the same notation as above for the constructors in the temporal formulae, without distinguishing the semantic translation.

Following common practice we simplify the description of proof and semantics for temporal formulae by distinguishing a "core" collection of the constructors, and interpreting the other operators in terms of this core. The core temporal constructors we adopt are **true**, $\wedge$, $\forall$, **pred**, **Previously**, and **FromThenOn**. The other operators are eliminated via the following equivalences.

$$\begin{array}{rcl}
\textbf{false} & \equiv & \textbf{Not true} \\
\textbf{Not Not}\ p & \equiv & p \\
p \vee q & \equiv & \textbf{Not}(\textbf{Not}\ p \wedge \textbf{Not}\ q) \\
p \Rightarrow q & \equiv & \textbf{Not}\ p \vee q \\
p \Leftrightarrow q & \equiv & (p \Rightarrow q) \wedge (q \Rightarrow p) \\
\exists b \bullet p(b) & \equiv & \textbf{Not}(\forall b \bullet \textbf{Not}\ p(b)) \\
\textbf{first} & \equiv & \textbf{Previously false} \\
\textbf{PreviouslyS}\ p & \equiv & \textbf{Not}(\textbf{Previously}(\textbf{Not}\ p)) \\
\textbf{Always}\ p & \equiv & \textbf{false FromThenOn}\ p \\
\textbf{Sometime}\ p & \equiv & \textbf{Not}(\textbf{Always}(\textbf{Not}\ p)) \\
\textbf{init}\ p & \equiv & \textbf{Always}(\textbf{first} \Rightarrow p) \\
\textbf{FromThenOnS}\ p\ q & \equiv & (\textbf{Sometime}\ p) \wedge (p\ \textbf{FromThenOn}\ q) \\
\textbf{MostRecently}\ p\ q & \equiv & (p \wedge q)\ \textbf{FromThenOn}\ (p \Rightarrow q) \\
\textbf{MostRecentlyS}\ p\ q & \equiv & (p \wedge q)\ \textbf{FromThenOnS}\ (p \Rightarrow q)
\end{array}$$

4.2.1 Semantics of temporal formulae

We introduce the following notation for treating histories, where ρ denotes a (possibly empty, finite) list of configurations.

- The history with elements a, b, c is represented by $\langle a, b, c\rangle$.
- The concatenation of ρ and history σ is denoted by $\rho^\frown\sigma$.
- The prefix relation on histories is denoted by $(\_ \preceq \_)$.
- The last ("current") element of a history σ is denoted by $\downarrow\!\sigma$; that is,

$$\downarrow\!(\tau^\frown\langle a\rangle) = a\,.$$

- The front of a history σ (its "previous history") is denoted by σ^-; that is,

$$(\rho^\frown\langle a\rangle)^- = \rho\,.$$

Note that all these operations are well-defined on histories except for "front", which only produces a history if taken on a history of length larger than one.

The notion that a given history satisfies a temporal formula is now captured as the statement that a history σ *models* the temporal formula p, written $\sigma \models p$, defined inductively by

- $\sigma \models$ **true** iff True;
- $\sigma \models$ **pred** b iff $b(\downarrow\sigma)$;
- $\sigma \models \mathsf{At}\ S$ iff for some t, m, i, $(\downarrow\sigma) = (S, t, , i, m)$;
- $\sigma \models \mathsf{By}\ T$ iff for some s, m, i, $(\downarrow\sigma) = (s, T, i, m)$;
- $\sigma \models$ **Not** q iff not $\sigma \models q$;
- $\sigma \models q \wedge r$ iff $\sigma \models q$ and $\sigma \models r$;
- $\sigma \models \forall x \bullet (q\ x)$ iff for all x, $\sigma \models (q\ x)$;
- $\sigma \models$ **Previously** q iff $\sigma^- = \langle\ \rangle$ or $\sigma^- \models q$;
- $\sigma \models q$ **FromThenOn** r iff either,
  - for all histories σ' with $\sigma' \preceq \sigma$, $\sigma' \models r$, or else,
  - there exists a history σ'' with $\sigma'' \preceq \sigma$ such that $\sigma'' \models q$ and for all histories σ' with $\sigma'' \preceq \sigma' \preceq \sigma$, $\sigma' \models r$.

We say that a formula p is *valid*, written $\models p$, if and only if for all histories σ, $\sigma \models p$.

In summary, let us reinterpret these definitions in terms of history predicates. Temporal formulae are predicates over histories, the temporal constructors generate temporal formulae from configuration properties. Thus, the statement $\sigma \models p$ is equivalent to $(p\ \sigma)$, the propositional constructors are just the obvious lifts, and validity is just entailment (from **true**).

4.2.2 Semantics of the turnstile

In reasoning about state machine properties we must restrict from all histories to executions obtained by the evolution of a given machine. Let M be a state machine. If σ is an *execution* of $\mathcal{M}$ we write $\sigma \models \mathcal{M}$. Then we say that a temporal formula p is $\mathcal{M}$-valid if and only if for all histories σ such that $\sigma \models \mathcal{M}$ (that is, all executions of $\mathcal{M}$), $\sigma \models p$.

The semantic definition of the sequent turnstile now proceeds similarly. The boolean $\Gamma \vdash p$ is true if and only if for every history σ such that $\sigma \models \mathcal{M}$, $\sigma \models p$ or else there is some q in Γ such that not $\sigma \models q$.

To be even more specific, let $\bigwedge \Gamma$ denote the temporal property which is the conjunction of all the hypotheses in Γ. Then $\Gamma \vdash p$ if and only if the temporal formula $((\bigwedge \Gamma) \Rightarrow p)$ is $\mathcal{M}$-valid. Thus, *verification* of a machine property p is simply demonstrating that p is $\mathcal{M}$-valid, or equivalently $\vdash p$.

As a final rewriting, it will be convenient in the discussion of sequent calculus to introduce the history predicate $\mathsf{Exec}_{\mathcal{M}}$ which characterizes executions; that is,

$$\mathsf{Exec}_{\mathcal{M}} = (\lambda\sigma \bullet \sigma \models \mathcal{M})\,.$$

Then, we have $\Gamma \vdash p$ – i.e., $((\bigwedge \Gamma) \Rightarrow p)$ is $\mathcal{M}$-valid – if and only if

$$(((\bigwedge \Gamma) \wedge \mathrm{Exec}_{\mathcal{M}}) \Rightarrow p) \text{ is valid.}$$

One could think of validity *without* the history predicate as a "pure temporal logic" turnstile. Although we will not introduce a separate syntax, we will refer to this as "reasoning at the abstract temporal logic level". Thus we see that the difference between reasoning at the level of the machine, and at the abstract temporal logic level, is manifestly just the inclusion of an additional hypothesis $\mathrm{Exec}_{\mathcal{M}}$ in all sequents. We will see that this is nontrivial in the next section.

Chapter 5

A calculational scheme for verifying machine properties

In this section we consider the proof system for verifying machine properties. It is presented in terms of sequent calculus, so the backbone is a set of structural rules for manipulating sequents, including the usual propositional calculus. The sequent calculus is ideal for encoding the structural reasoning in an immediate manner which aids the proof visualization which the DOVE tool is striving for.

However, it will have already been obvious from the semantics in the previous section that reasoning with temporal operators such as **FromThenOn** is fraught with difficulties. The definitions are not always intuitive, and temporal notions are notoriously ambiguous and context dependent. Thus, although one can extend the sequent calculus to include temporal operator introduction and elimination, and even detailed induction rules, this is not the focus of the DOVE proof system.

Instead, in DOVE an iterative calculational scheme is employed which – loosely speaking – drags the proof burden back to initial conditions by "backsubstituting" the property through the transitions of the machine. All temporal operators are then dealt with through this induction.

The notion of backsubstitution, and an outline of the calculational scheme, are presented below, after a discussion of the sequent calculus. The implementation in the current version of DOVE is discussed in the next section.

5.1 A calculus for manipulating sequents

Proving a machine property usually involves a combination of temporal and non-temporal reasoning. For non-temporal reasoning, we need two kinds of rule: *structural rules*, which specify how one may combine, add to, delete from, and rearrange properties within a sequent; and non-temporal operator *introduction rules* – further classified into *left* and *right introduction rules* for $\wedge, \neg, \vee\Rightarrow, \forall$ and $\exists$ – which are used for reconstructing a sequent in reasoning about properties of a machine.

For temporal reasoning, there are similar temporal operator introduction and elimination rules, as well as *induction rules* which allow reasoning about the temporal operators **FromThenOn** and **Always** inductively over histories. As discussed above the former are not crucial in the DOVE proof system, and from the latter we will just use forwards induction for **Always** as arises naturally in the calculational scheme presented in the next section.

All of these axioms and rules are straightforward to prove in the semantics of the previous section. That is, their *soundness* may be demonstrated straightforwardly, and we omit the proofs which are somewhat tedious.

5.1.1 Non-temporal sequent calculus in DOVE

The variable conventions used in the following are:

$p, q, r, \ldots, p_1, p_2, \ldots$	history properties
Γ, Δ	hypothesis lists
$\mathbf{Always}\Gamma$	distributed operator
$x, y, \ldots$	variables
$x: R$	a variable x with the type R
$a, b \ldots$	constants

5.1.1.1 Axioms

$$\frac{}{\Gamma, p \vdash p}(basic) \qquad \frac{}{\Gamma \vdash \mathbf{true}}(\top)$$

$$\frac{}{\Gamma \vdash \mathbf{pred}\, p}(\uparrow p) \quad \text{all } p \text{ such that for all } c,\ p\ c$$

5.1.1.2 Inference rules for non-temporal reasoning

Non-temporal inference rules can be classified into two classes as follows.

(1) ***Structural rules***

$$\frac{\Gamma \vdash q}{\Gamma, p \vdash q}(ext) \quad \frac{\Gamma_1, p_1, \Gamma_2, p_2, \Gamma_3 \vdash q}{\Gamma_1, p_2, \Gamma_2, p_1, \Gamma_3 \vdash q}(com)$$

$$\frac{\Gamma, p, p \vdash q}{\Gamma, p \vdash q}(simp) \quad \frac{\Gamma, p \vdash q \qquad \Delta \vdash p}{\Gamma, \Delta \vdash q}(cut)$$

(2) ***Non-temporal operator introduction rules***

$$\frac{\Gamma, p_1, p_2 \vdash q}{\Gamma, p_1 \wedge p_2 \vdash q}(\wedge_l) \quad \frac{\Gamma \vdash q_1 \qquad \Delta \vdash q_2}{\Gamma, \Delta \vdash q_1 \wedge q_2}(\wedge_r)$$

$$\frac{\Gamma \vdash q}{\Gamma, \mathbf{Not} q \vdash \mathbf{false}}(\mathbf{Not}_l) \quad \frac{\Gamma, p \vdash \mathbf{false}}{\Gamma \vdash \mathbf{Not} p}(\mathbf{Not}_r)$$

$$\frac{\Gamma \vdash p_1 \qquad \Gamma, p_2 \vdash q}{\Gamma, p_1 \Rightarrow p_2 \vdash q}(\Rightarrow_l) \quad \frac{\Gamma, p \vdash q}{\Gamma \vdash p \Rightarrow q}(\Rightarrow_r)$$

$$\frac{\Gamma, p(y) \vdash q}{\Gamma, \forall x \bullet p(x) \vdash q}(\forall_l) \quad \frac{\Gamma \vdash q(y)}{\Gamma \vdash \forall x \bullet q(x)}(\forall_r)$$

Note that all the above rules hold even before restriction to executions – i.e., at the abstract temporal logic level in the sense of the discussion at the end of Chapter 4 – as is manifest since in each rule we could just replace hypotheses[3] Γ with $\Gamma, \mathrm{Exec}_{\mathcal{M}}$.

From these rules and the temporal equivalences listed in Section 4.2 we can clearly derive similar introduction rules for the remaining predicate logic constructors. We will later require the following rule for removing disjunction on the left of the turnstile,

$$\frac{\Gamma, p_1 \vdash q \qquad \Gamma, p_2 \vdash q}{\Gamma, p_1 \vee p_2 \vdash q}(\vee_l)$$

5.1.2 On the temporal extension and further discussion

The only general temporal reasoning rule that we will add to the DOVE sequent calculus is the following forward induction rule.

$$\frac{\mathbf{init}\,\Gamma_1, \mathbf{Always}\ \Gamma_2, \mathbf{Previously}(\mathbf{Always}\ r) \vdash r}{\mathbf{init}\,\Gamma_1, \mathbf{Always}\ \Gamma_2 \vdash \mathbf{Always}\ r}(F_{\mathbf{Always}})$$

The more usual "initial and inductive cases" form is obtained by decomposing under **first** or **Notfirst**. If **first**, then **Previously**(**Always** r) can be replaced by **true**, as is obvious from the semantics in Section 4.2.1.

Even this rule holds at the abstract temporal logic level. The crucial observation is that $\mathrm{Exec}_{\mathcal{M}} = (\mathbf{Always}\ \mathrm{Exec}_{\mathcal{M}})$, as is clear from the semantics in Chapter 4 – executions are iteratively built up from executions.

Induction rules for the temporal operator **FromThenOn** can also be written down, as well as introduction and elimination rules. Similarly, we can write an introduction rule for **Previously**. However, the elimination rule is problematic. For, at the abstract temporal logic level the elimination rule

$$\frac{\mathbf{Always}\ \Gamma_1, \mathbf{Previously}\ \Gamma_2 \vdash \mathbf{Previously}\ q}{\mathbf{Always}\ \Gamma_1, \Gamma_2 \vdash q}$$

[3] Where appropriate, also replace Δ with $\Delta, \mathrm{Exec}_{\mathcal{M}}$ and then use the rule *ext*.

fails the semantic translation, as is easily checked. The alternative rule

$$\frac{\textbf{Previously}\ \Gamma \vdash \textbf{Previously}\ q}{\Gamma \vdash q}$$

is sound at the abstract temporal logic level, but clearly has no extension to the machine level. Thus, reasoning based only on sequent calculus is likely to be incomplete, beyond all the problems of intuition mentioned earlier.

Fortunately, an alternative proof process – to which we turn next – can be applied quite generally.

5.2 The iterative proof scheme

In this section we outline the scheme in a general setting, beginning with the notion of predicate transformer embedding of the transition system.

5.2.1 Predicate transformers

We have already seen, in Section 3.3, that each edge of a given state-machine is associated to a relation between configurations. The state machine can then be identified with a transition system with relation semantics. For calculational convenience, we embed this description in a predicate transformer model.

Configuration predicate transformers are simply mappings in $(\mathsf{Config}_{\mathcal{M}} \rightarrow \mathbb{B}) \rightarrow (\mathsf{Config}_{\mathcal{M}} \rightarrow \mathbb{B})$. A given relation r between configurations is embedded "demonically" in this model as the predicate transformer $\mathsf{wp}\ r$, such that

$$(\mathsf{wp}\ r)\ \phi\ c = (\forall c' \bullet (r\ c\ c') \Rightarrow (\phi\ c'))\,.$$

Clearly, $(\mathsf{wp}\ r)\ \phi$ is the *weakest precondition* to establish the configuration predicate ϕ via the "evolution" corresponding to the relation r. The calculational convenience of representing transitions as mappings between predicates follows from the fact that these predicates are just properties of interest in the machine. More generally it is temporal formulae – history predicates – which are of interest. Representing the evolution of the system directly on temporal formulae is obviously useful for the problem of verification. So, we must lift the notion of predicate transformer to history predicates.

There are a number of ways of constructing the resulting "temporal" predicate transformers, but probably the most direct is simply to first extend the transition relation to a relation between histories[4], and take the demonic embedding of that as before. Thereto, we define the temporal relation r_{T} corresponding to configuration relation r by

$$r_{\mathrm{T}}\ \sigma\ \sigma' \ = \begin{cases} \mathsf{false}\,, \ \text{if } \sigma' = \langle c' \rangle \text{ for some configuration } c'\,, \\ (\sigma = \sigma'^{-})\ \&\ (r\downarrow\!\sigma\ \ \downarrow\!\sigma')\,, \ \text{otherwise.} \end{cases}$$

[4] Recall that a history is a *non-empty* list of configurations.

The temporal predicate transformer (wtp r) is then obtained by demonic embedding,

$$(\mathsf{wtp}\ r)\ p\ \sigma = \forall \sigma' \bullet (r_{\mathrm{T}}\ \sigma\ \sigma') \Rightarrow (p\ \sigma') \,.$$

The benefit of this straightforward approach is that (wtp r) p is manifestly the *weakest temporal precondition* to establish a temporal property p by executing the "program" embodied in the relation r_{T}. Inserting the definition of r_{T} leaves the alternate form

$$(\mathsf{wtp}\ r)\ p\ \sigma = (\mathsf{wp}\ r)\ (\lambda c \bullet p\ \sigma^\frown\langle c\rangle)\ \downarrow\sigma \,,$$

which emphasizes the fact that the temporal predicate transformer is built up "locally" by simply extending the history according to the relation r.

5.2.2 Predicate transformer embedding of the transition system

The predicate transformer model for a given state-machine $\mathcal{M}$ now follows by embedding the transition relations. The temporal relation corresponding to an edge ϵ, $r = \rho_\epsilon$, relates a given history to the history which extends it consistently with the connecting transition in the given edge. In this way, the transition system may be extended to a temporal transition system. The initial condition becomes

$$\iota_{\mathrm{T}} = (\mathbf{first}\ \wedge\ \mathbf{pred}(\lambda c\ \bullet \iota_{\mathcal{M}}\ c)) \,,$$

and the total transition relation is extended as above. We will denote by $\mathcal{T}_{\mathrm{T}}$ the total predicate transformer so obtained,

$$\mathcal{T}_{\mathrm{T}} = (\mathsf{wtp}\ \mathcal{R}_{\mathcal{M}}) \,.$$

This will be used extensively in the following. $\mathcal{T}_{\mathrm{T}}\ p$ computes the *weakest temporal precondition* to establish a temporal property p in the next step in the evolution.

We will also use the statement that a given temporal formula is true in all initial configurations of the given machine, which can be represented via a mapping $\mathcal{I}_{\mathrm{T}}$ from temporal formulae to $\mathbb{B}$,

$$\mathcal{I}_{\mathrm{T}}\ p = (\mathbf{first} \vdash p) \,.$$

5.2.3 Calculation of the wtp in DOVE state-machines

For any *configuration* predicate ϕ, and configuration relation r, the definitions in Section 5.2.1 immediately give

$$(\mathsf{wtp}\ r)\ \ \mathbf{pred}(\lambda c \bullet \phi\ c) = \mathbf{pred}(\lambda c \bullet (\mathsf{wp}\ r)\ \phi\ c) \,.$$

If we had nice intertwining properties for wtp with the temporal logic constructors then an efficient computation of a given temporal predicate transformer on an arbitrary temporal formula would be to intertwine through to the base configuration properties in the formula.

However, nice intertwining properties only exist when r is restricted to be deterministic – i.e., $r = \widehat{f}$, where

$$\widehat{f}\ c\ c' = (c' = (f\ c))$$

for some function f – in which case, clearly

$$(\mathsf{wp}\ \widehat{f})\ \phi\ c = (\phi\ (f\ c))\,.$$

The intertwinings with the remaining constructors of the temporal logic are then reasonably intuitive for the deterministic relation.

$$\begin{array}{rcl}
(\mathsf{wtp}\ \widehat{f})\ (\textbf{Not}\ p) & = & \textbf{Not}((\mathsf{wtp}\ \widehat{f})\ p) \\
(\mathsf{wtp}\ \widehat{f})\ (p \wedge q) & = & ((\mathsf{wtp}\ \widehat{f})\ p) \wedge ((\mathsf{wtp}\ \widehat{f})\ q) \\
(\mathsf{wtp}\ \widehat{f})\ (\forall x \bullet p\ x) & = & \forall x \bullet (\mathsf{wtp}\ \widehat{f})\ (p\ x) \\
(\mathsf{wtp}\ \widehat{f})\ (\textbf{Previously}\ p) & = & p \\
(\mathsf{wtp}\ \widehat{f})\ (p\ \textbf{FromThenOn}\ q) & = & ((\mathsf{wtp}\ \widehat{f})\ q) \wedge (((\mathsf{wtp}\ \widehat{f})\ p) \vee \\
 & & (p\ \textbf{FromThenOn}\ q))
\end{array}$$

There are two reasons – manifest in Definition 3 of Section 3.2 – why this does not hold for the transition relations derived from the transition definitions in a DOVE state machine: Firstly, the assignments are guarded. Secondly, the inputs are not determined. However, as discussed there, we have explicitly separated out a deterministic assignment which therefore *will* have nice intertwining properties. Inserting that form into the demonic embedding formula, and keeping the input quantification explicit, we are left – for a given edge ϵ – with

$$(\mathsf{wtp}\ \rho_\epsilon)\ p\ \sigma = (\forall i \bullet \mathcal{G}_\epsilon \downarrow\!\sigma \Rightarrow (\mathsf{wtp}\ \widehat{\mathcal{F}^i_\epsilon})\ p\ \sigma)\,.$$

We will use the shorthand $\mathcal{A}^i_\epsilon$ for $(\mathsf{wtp}\ \widehat{\mathcal{F}^i_\epsilon})$, the temporal predicate transformer corresponding to the deterministic assignment; thus, we have

$$\mathcal{A}^i_\epsilon\ p\ \sigma = (p\ \sigma\frown\langle\mathcal{F}^i_\epsilon \downarrow\!\sigma\rangle)\,.$$

We will refer to the temporal formula $(\mathcal{A}^i_\epsilon\ p)$ as the *back-substitution of p through edge ϵ* (while updating the inputs with value i). This is the form which will be utilised in the DOVE implementation.

5.2.4 The iterative calculation

The aim now is to outline informally the general scheme for verifying temporal formulae.

Recall that $\mathcal{T}_\mathrm{T}\ p$ computes the weakest temporal precondition that p is established at the next step in the evolution. Thus the n^{th} composition, which we denote $\mathcal{T}^n_\mathrm{T}\ p$ computes the weakest temporal precondition that p is established after n more steps in the evolution. From this composition starting in the initial configuration, we obtain the statement

$$\mathcal{I}_\mathrm{T}(\mathcal{T}^n_\mathrm{T}\ p)\,,$$

that all executions of length n model p. Finally then, we have that

$$\vdash p = (\forall n \bullet \mathcal{I}_\mathrm{T}(\mathcal{T}^n_\mathrm{T}\ p)\,.$$

We will refer to this as the *composite representation* of $\mathcal{M}$-validity. Again, this just embodies the obvious statement that verification of a machine property p can be split up into showing that all executions of length n model p, for all n.

Of course this is still an infinite number of requirements, so not much progress is apparent! However, recall the notion of an *invariant*. We say that p is an invariant of a given machine $\mathcal{M}$ if:

1. $\mathcal{I}_{\mathrm{T}}\ p$; and,
2. $p \Rightarrow (\mathcal{T}_{\mathrm{T}}\ p)$ is valid.

In particular, this is invariance under a step in the evolution of the machine. Invariance implies $\mathcal{M}$-validity: 2 effectively says that if a given execution models p, so does its extension by one step, and 1 says an execution of length one models p. Moreover, to check invariance is a problem in sequent calculus which is rather simpler than proving the full $\mathcal{M}$-validity in general.

The idea of the iterative calculation may now be apparent. First we check whether the first n extensions from any initial configuration satisfy p. We then generalize the "inductive" part of the invariance definition to check if, for any history, the extension by $n+1$ steps models p when we know extensions up to the n^{th} step do. This can be carried out successively in n, until the inductive step is successful, or the "initial case" fails. The following is a high-level "algorithm" for verifying a machine property p.

Step 1. Initialize Acc to p, and initialize Pres to p.

Step 2. Check $\mathcal{I}_{\mathrm{T}}$ Pres.

Step 3. If this fails then finish with result False, otherwise

Step 4. check that the temporal formula (Acc $\Rightarrow$ ($\mathcal{T}_{\mathrm{T}}$ Pres) is valid. In practice, this means to apply sequent calculus at the abstract temporal logic level.

Step 5. If this succeeds then finish with result True, otherwise

Step 6. reset Acc to (Acc $\wedge$ ($\mathcal{T}_{\mathrm{T}}$ Pres)), and reset Pres to ($\mathcal{T}_{\mathrm{T}}$ Pres), and proceed from Step 2.

The result of the iteration terminating is either that a temporal formula p is falsified in a finite number of steps, or that for some $n \geq 0$

$$\mathcal{I}_{\mathrm{T}}\ p\,,\ \ \mathcal{I}_{\mathrm{T}}\ (\mathcal{T}_{\mathrm{T}}\ p)\,, \ldots, \mathcal{I}_{\mathrm{T}}\ (\mathcal{T}_{\mathrm{T}}^{n}\ p)\,,$$

and we have established the validity of

$$p\ \wedge\ (\mathcal{T}_{\mathrm{T}}\ p)\ \wedge\ \ldots \wedge\ (\mathcal{T}_{\mathrm{T}}^{n}\ p) \Rightarrow (\mathcal{T}_{\mathrm{T}}^{n+1}\ p)\,.$$

These last two results together clearly imply the validity of p.

An alternative statement of the successful termination is that, for some $n \geq 0$, the property $p\ \wedge\ (\mathcal{T}_{\mathrm{T}}\ p)\ \wedge\ \ldots \wedge\ (\mathcal{T}_{\mathrm{T}}^{n}\ p)$ is an *invariant* of the state-machine (and, in particular, is $\mathcal{M}$-valid). From a straightforward argument using the *cut* rule we then immediately deduce $\mathcal{M}$-validity of p.

Chapter 6

Implementation of the proof scheme in DOVE

The DOVE implementation of the calculational scheme introduced in Section 5.2.4 must respect the fact that the user needs to be comfortable with the information being presented at each step. It must also take into account that many users are not particularly familiar with the detailed use of formal methods for verification.

In Step 4 of the algorithm, sequent calculus must be used judiciously to prove validity of a given formula which is apparently diverging further and further from the original goal. The user must make a decision, at each iteration, as to whether this can be completed successfully, or whether to return to the branching goal and proceed with further iteration of the algorithm. This is problematic, both because the decision may be quite nonobvious, and because the work done in trying to verify the formula is then completely wasted.

In the current version of DOVE a modified scheme has been pursued. It is "morally" based on the proof scheme of Section 5.2.4, but differs in a number of respects.

- Previously, the full transition relation was used at all steps. This is good for keeping all hypotheses constructed from different transitions together, but makes for quite a mess of information. In the modified scheme the goal to be proved is first decomposed into subgoals under the topology of the graph, and the weakest temporal precondition in a given subgoal is then only taken with respect to the corresponding edge.
- Previously, the iteration formula at Step 4 must be shown valid – i.e., verified at the abstract temporal logic level. This may allow a stronger sequent calculus, but took the user into a different context. In the modified scheme the reasoning is, at all stages, at the machine level.
- Previously, if the sequent calculus did not terminate the algorithm at Step 4, the user must return to a previous stage – thus "losing" all that effort – before applying the next iteration of the algorithm. In the modified scheme the user launches a new iteration from the proof state remaining after the sequent calculus verification attempt.

In this way the implementation desiderata are reasonably well achieved. The cost is that the completeness of the scheme is no longer obvious.

In this section we provide details of the modified scheme and its corresponding DOVE support. A comparison to the complete scheme in Section 5.2 is provided. We end with an illustration of its use in the TrafficLights theory.

6.1 Implementing a modified proof scheme

The problem at hand is to verify a given temporal formula in a given state machine. To do this, we apply the following strategy.

Tactic 1. Induction: apply the temporal induction rule of Section 5.1.2.

Tactic 2. Topology: separate the initial case from the goal to be proved, and solve it using the initial predicate information and the (non-temporal) sequent calculus of Section 5.1.1. If this fails then the property is false and the strategy terminates. Otherwise, further decompose into a set of subgoals characterized by the edges of the state machine graph.

Tactic 4. Back-Substitute: back-substitute a given subgoal through the corresponding edge to drag the proof burden back to the context of the previous state.

Tactic 5. Sequent Calculus: apply the (non-temporal) sequent calculus of Section 5.1.1 and attempt to discharge the resulting subgoals.

Iteration: If unsuccessful, the user can now continue iterating this procedure from Tactic 2. Otherwise, the user can start on another subgoal.

When all subgoals have been verified, the original goal has been proved.

Note that the tactic Sequent Calculus has been covered in Section 5.1, so we concentrate on the other tactics of the modified scheme

6.1.1 Implementing Induction

The temporal induction rule has also been presented in Section 5.1. One subtlety, whose relevance will be seen below, is that the first thing to do in this step – i.e., before applying the induction rule – is to apply the rule

$$\frac{\vdash \textbf{Always}\ p}{\vdash p}$$

In fact, there is a temporal equivalence that $\vdash p$ if an only if $\vdash$ **Always** p. The former says that all executions model p, whilst the latter says that all prefixes of all executions model p – obviously these are semantically the same thing. If p is already an "always" property, then this rule has no effect.

6.1.2 Implementing Topology

This tactic is implemented via a number of obvious theorems which simply encode the topology of the state machine. If **Not first** holds at some point in any execution, then the corresponding configuration must record the end state of an edge in the state machine graph. Thus we have

$$\frac{}{\vdash \mathbf{first} \vee (\bigvee_{b \underline{t} e} \mathbf{Not\ first} \wedge \mathbf{Previously}(\mathsf{At}\ b)\ \wedge\ \mathsf{By}\ t\ \wedge\ \mathsf{At}\ e)}(\mathsf{EC})\,.$$

This theorem may be applied to decompose the current proof state into a number of subgoals: one subgoal for the initial case, and one for each edge in the state machine graph. We call these the initial case decomposition, and the edge decomposition, respectively.

To achieve the decompositions, consider a sequent representing the proof state, or *goal*, $\Gamma \vdash p$. We can now use the sequent calculus rule *cut*, resolved with EC, to introduce the further hypothesis $\mathbf{first} \vee (\bigvee_{b \underline{t} e} \mathbf{Not\ first} \wedge \mathbf{Previously}(\mathsf{At}\ b)\ \wedge\ \mathsf{By}\ t\ \wedge\ \mathsf{At}\ e)$. We say that the hypothesis has been *cut into* the sequent. The resulting sequent is now decomposed into two by apply the rule $\vee_l$. This is the initial case decomposition. The initial state information can be cut into the **first** subgoal via the following rule

$$\frac{}{\mathbf{first} \vdash \mathsf{At\ InitialState}}(\mathsf{IS})\,.$$

Repeated application of $\vee_l$ on the remaining subgoal then effects the edge decomposition.

Each subgoal now refers to a specific part of the state machine, and this information is reported to the user by highlighting of the corresponding edge of state on the state machine graph. This immediate proof visualization can be useful for book-keeping in a complicated proof.

6.1.3 Implementing Back-Substitute

A sequent representing a subgoal in the proof state at some point in the proof procedure can be verified via its weakest temporal precondition. For, observe that

$$\frac{\mathbf{first} \vdash p \qquad \vdash \mathcal{T}_{\mathrm{T}}\ p}{\vdash p},$$

which is a simple consequence of the composite representation of $\mathcal{M}$-validity, and the definition of $\mathcal{I}_{\mathrm{T}}$. Of course, a given subgoal will also have hypotheses, so we actually must apply this result with p replaced by $\Gamma \Rightarrow p$, giving

$$\frac{\mathbf{first}, \Gamma \vdash p \qquad \vdash \mathcal{T}_{\mathrm{T}}(\Gamma \Rightarrow p)}{\Gamma \vdash p}.$$

To simplify this expression we first observe that, as a consequence of the Topology tactic, there will be a unique initial case. The Back-Substitute tactic cuts the initial condition into this case via the obvious theorem

$$\frac{}{\mathbf{first} \vdash \mathsf{InitialPredicate}}(\mathsf{IP})\,,$$

and then simplifies (or, if possible, discharges) it using sequent calculus.

The subgoals remaing from the Topology tactic refer to a fixed edge with the hypothesis **Not first**. We may then use the guarded action decomposition of Section 5.2.3 for back-substitution through the given edge. The desired rule for $b \overset{t}{-} e$ is then easily derived to be

$$\frac{\mathcal{G}_{b\overset{t}{-}e}, \mathcal{A}^i_{b\overset{t}{-}e}\ \Gamma \vdash \mathcal{A}^i_{b\overset{t}{-}e}\ p}{\textbf{Not first}, \textbf{Previously}(\mathsf{At}\ b), \mathsf{By}\ t, \mathsf{At}\ e, \Gamma \vdash p}(BS_{bte})$$

Note that DOVE pretty-prints the semantics back into concrete syntax for display to the user. Here we follow this more accessible form.

The Back-Substitute tactic applies this rule to a given subgoal in the edge decomposition. It again continues with sequent calculus rules in an attempt to discharge the resulting subgoal. It is worth stressing that the back-substitution rule for edge $b \overset{t}{-} e$, can only be applied *after* the Topology tactic.

6.2 Comparison of the proof schemes

The connection of the modified scheme to the complete scheme is not at all immediately obvious. However, some motivating remarks may be useful.

Recall that in the complete scheme, for each $n \geq 0$, we are testing

$$\textbf{first} \vdash p\,, \quad \textbf{first} \vdash (\mathcal{T}_\mathrm{T}\ p)\,, \quad \ldots \quad \textbf{first} \vdash (\mathcal{T}^n_\mathrm{T}\ p)\,,$$

and the validity of

$$p\ \wedge\ (\mathcal{T}_\mathrm{T}\ p)\ \wedge\ \ldots\ \wedge\ (\mathcal{T}^n_\mathrm{T}\ p) \Rightarrow (\mathcal{T}^{n+1}_\mathrm{T}\ p)\,.$$

Observe that we can replace the latter formula with

$$p,\ (\mathcal{T}_\mathrm{T}\ p),\ \ldots, (\mathcal{T}^n_\mathrm{T}\ p) \vdash (\mathcal{T}^{n+1}_\mathrm{T}\ p)\,,$$

without any loss of generality (though possible loss of efficiency) since $\mathcal{T}_\mathrm{T}$ extends executions to be executions. We will now see how such formulae appear at intermediate stages of the modified scheme, albeit with $\mathcal{T}_\mathrm{T}$ broken up edge-wise as a result of the topology tactic.

In the modified scheme, to verify p we start with $\vdash p$ and apply the induction rule $F_{\textbf{Always}}$. The resulting goal is

$$\textbf{Previously}(\textbf{Always}\ p) \vdash p\,.$$

Applying the Topology tactic then produces the initial case **first** $\vdash p$, and the edge decomposition of **Not first**, **Previously**(**Always** p) $\vdash p$. We now use the back-substitution rule for each edge. From the intertwining rules of Section 5.2.3, for the deterministic action through edge ϵ we have

$$\begin{aligned} \mathcal{A}^i_\epsilon\ (\textbf{Not first}) &= \textbf{Not}\ (\mathcal{A}^i_\epsilon\ \textbf{first}) \\ &= \textbf{true}\,; \text{and,} \\ \\ \mathcal{A}^i_\epsilon\ \textbf{Previously}(\textbf{Always}\ p) &= \textbf{Always}\ p \end{aligned}$$

and the resulting subgoal for the given edge is

$$\mathcal{G}_\epsilon, \textbf{Always}\ p \vdash \mathcal{A}^i_\epsilon\ p\,.$$

At this stage we have, in essence, reproduced the first iteration of the complete scheme.

A further iteration can now be launched by applying the Topology tactic. The initial cases are **first** $\vdash \mathcal{A}^i_\epsilon\ p$ for each edge. Again, back-subsitution follows the further edge-wise decomposition .

It is now clear why we needed to replace p with **Always** p to start the modified scheme. The "**Previously**(**Always** p)" term brought into the hypotheses will generate edge-wise decompositions of the hypotheses "$(\mathcal{T}^n_{\mathrm{T}}\ p)$" obtained previously. For, using

$$\textbf{Always}\ p = (p\ \wedge\ \textbf{Previously}(\textbf{Always}\ p))$$

and the intertwining

$$\mathcal{A}^i_\epsilon\ (p \wedge q) = ((\mathcal{A}^i_\epsilon\ p) \wedge (\mathcal{A}^i_\epsilon\ q))$$

in the iterated edge-wise decomposition, this term back-substitutes from

$$p\ \wedge\ \textbf{Previously}(\textbf{Always}\ p)$$

at the first stage above, to

$$p\ \wedge\ (\mathcal{A}^i_{\epsilon'}\ p)\ \wedge\ \textbf{Previously}(\textbf{Always}\ p)$$

after the next stage. This pattern continues, each time producing an extra composition of back-substitution through an edge.

In this way, we are again essentially producing each iteration of the complete scheme (the appropriate initial cases produced correctly by the Topology tactic required to launch the iteration). Figure 6.1 illustrates this situation in an example where the state machine graph would have two edges. The left represents the complete scheme, and on the right we see how the edgewise paths are broken up in the intervening Topology tactic. Note that different paths on the right will in general be different lengths, depending for example on how far the user pursues Back-Substitute before being able to discharge the goal.

The difference between the two schemes is that, since $\mathcal{T}_{\mathrm{T}}$ is broken up edge-wise, the modified scheme at any given stage in the iteration will typically be missing hypotheses which are naturally included in the complete scheme. As a consequence, the user may have to prove intermediate lemmas – to be cut into a problematic subgoal – which produce the same effect as the missing hypotheses. This problem will only appear when back-substituting around loops in the state machine diagram – the simplest such being the "Wait" transitions in the traffic lights example.

6.3 An example proof in the TrafficLights theory

We consider the problem of verifying the property

$$\textbf{Always}((\textbf{Not first} \wedge \mathsf{By\ NSChangeGreen}) \Rightarrow \mathsf{NLight} = \mathsf{Green}).$$

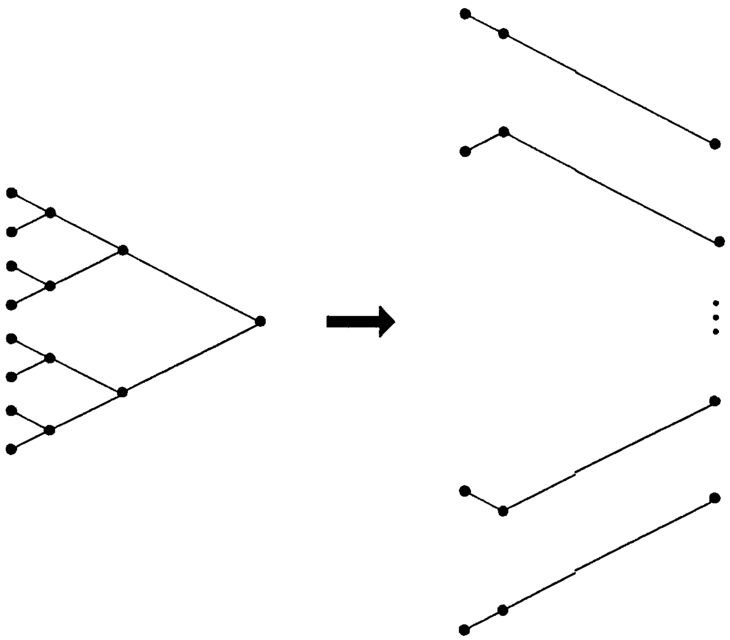

Figure 6.1: Comparison of proof schemes.

This is rather easy of course, and just presented as an illustration of the technique. The discussion is reasonably self-contained. In particular, the reader should observe that the only relevant edge in the first iteration of the proof strategy is (AllRed, NSChangeGreen, NSGreen).

The first step is to use the ($F_{\mathbf{Always}}$) rule to eliminate the **Always** operator by forward induction. Simplifying the result via application of ($\Rightarrow_r$) and ($\wedge_l$), leaves

Previously(**Always**((**Not first** ∧ By NSChangeGreen) ⇒ NLight = Green)),
Not first,
By NSChangeGreen
⊢ NLight = Green

The Topology tactic then separates out the initial case – which is trivial due to the **Not first** hypothesis – and leaves the single edge contribution

Not first,
Previously(**Always**((**Not first** ∧ By NSChangeGreen) ⇒ NLight = Green)),
Previously(At AllRed),
By NSChangeGreen,
At NSGreen
⊢ NLight = Green

The back-substitution rule BS is now applied, together with the intertwining rules for simplifying wtp, leaving the subgoal

(**Not first** ∧ By NSChangeGreen)⇒NLight = Green,
Previously(**Always**((**Not first** ∧ By NSChangeGreen) ⇒ NLight = Green)),
At AllRed
⊢ Green = Green

which is easily discharged.

6.3.1 Introducing intermediate lemmas

A slightly harder example, which demonstrates the necessity of introducing intermediate lemmas in the modified scheme, is verification of the property that specifies the colour of NLight in the state $\mathsf{EWAmber}$,

$$\vdash \mathbf{Always}(\mathbf{At}\ \mathsf{EWAmber} \Rightarrow \mathsf{NLight} = \mathsf{Red})\,.$$

The steps of the proof scheme are straightforward to carry out. The first step is to apply $(F_{\mathbf{Always}})$ and simplify, leaving

$$\begin{array}{l}\mathbf{Previously}(\mathbf{Always}(\mathbf{At}\ \mathsf{EWAmber} \Rightarrow \mathsf{NLight} = \mathsf{Red})),\\ \mathbf{At}\ \mathsf{EWAmber}\\ \vdash \mathsf{NLight} = \mathsf{Red}\end{array}$$

Applying the Topology tactic leaves the initial case – which is a simple contradiction between the hypothesis $\mathbf{At}\mathsf{EWAmber}$ and the fact that the initial state is AllRed – and the edge decomposition contributions from WaitTO and $\mathsf{EWChangeAmber}$. The wait transition subgoal is immediately discharged by the Back-Substitute tactic, since the **Previously** hypothesis comes into play when dragged back to the state $\mathsf{EWAmber}$.

Back-substitution through the remaining edge drags us back to the subgoal

$$\mathbf{At}\ \mathsf{EWGreen} \vdash \mathsf{NLight} = \mathsf{Red}\,.$$

Note that the "useless" hypotheses – for example those stating the colour of NLight in the state $\mathsf{EWAmber}$, and the rest of the transition guard – have been pruned away by using the tactic *ext*. At this stage another round of iteration is launched, beginning with the Topology tactic. The initial case is easy as before, leaving two cases:

- **By** $\mathsf{EWChangeGreen}$, which is dragged back under the Back-Substitute tactic to the state AllRed. That the North light is Red in AllRed is manifest when $\mathsf{LastGreen} = \mathsf{NS}$, as is implied by the guard of the transition $\mathsf{EWChangeGreen}$ and therefore appears as a hypothesis after backsubstitution. For, the back-substitution along $\mathsf{NSChangeRed}$, and the initial predicate applicable for the **first** case in AllRed, obviously impose $\mathsf{NLight} = \mathsf{Red}$, while back-substitution along $\mathsf{EWChangeRed}$ requires $\mathsf{LastGreen} = \mathsf{EW}$ and so produces contradictory hypotheses.

- **By** WaitNS, for which the corresponding subgoal has the form

$$\begin{array}{l}\mathbf{Not\ first},\\ \mathbf{Previously}(\mathbf{Always}(\mathbf{At}\ \mathsf{EWAmber} \Rightarrow \mathsf{NLight} = \mathsf{Red})),\\ (\mathbf{At}\ \mathsf{EWAmber} \Rightarrow \mathsf{NLight} = \mathsf{Red}),\\ \mathbf{Previously}(\mathbf{At}\ \mathsf{EWGreen}),\\ \mathbf{By}\ \mathsf{WaitNS},\\ \mathbf{At}\ \mathsf{EWGreen}\\ \vdash \mathsf{NLight} = \mathsf{Red}\,.\end{array}$$

> Under the Back-Substitute tactic the state returns to EWGreen. However, the hypotheses about the state EWAmber are clearly of no use in proving something about EWGreen. Hence, this subgoal cannot be directly proved by back-substitution. However, the fact that NLight is Red in the state EWGreen is manifestly correct from the definition of the TrafficLights state machine!

Thus this second item brings up the required lemma, which is simply the property which specifies that NLight is also Red in the state EWGreen,

$$\vdash \textbf{Always}(\textbf{At}\ \mathsf{EWGreen} \Rightarrow \mathsf{NLight} = \mathsf{Red})\,.$$

Chapter 7

Conclusions

This paper has presented the foundations of a formal-theoretic approach to state machine designs in the DOVE system. Specifically, we have provided a language for representing behavioural properties of state machines, and proposed a formal proof system for verifying that proposed properties are satisfied by given designs. From this formal structure, a detailed discussion of the theoretical aspects of the DOVE system has been given.

The DOVE system has been implemented and released. Its implementation not only provides a powerful tool for the design of critical systems, but also demonstrates how domain-specific proof steps – such as repeated back-substitution – can be compiled into tactics for use in a general-purpose, low-level proof assistant such as Isabelle.

In the remainder of this last section we present our outlook for future work, after a quick review of the related literature.

7.1 Related work

As is well known, temporal logic has been widely used as a formalism for program specification and verification [8, 17, 24, 25], temporal reasoning [36, 38], modeling temporal databases [4, 14, 15, 30], simulation applications [21, 41] and so on. More recently, there has been a substantial interest in the use of temporal logic for specifying properties of reactive systems [5, 7, 12, 18, 22, 25, 26, 27]. For comparison with the DOVE system, we will briefly consider just two from this list: the logic of Manna and Pnueli [25, 27]; and TLA (the Temporal Logic of Actions), proposed by Lamport [18].

In the logic of Manna and Pnueli, transition systems [9] are used as a computational model for reactive systems, and temporal logic is used as a specification language to express properties which must be satisfied by any proposed implementation.

The dual language approach employed in the DOVE system is similar to the approach proposed by Manna and Pnueli [27], which the STeP (Stanford Temporal Prover) system [3] that supports the computer-aided verification of reactive systems is based on. As in the DOVE system, the language for specifying state-machines is based on state transition diagrams and the formal meaning of a state machine is determined in terms of sequences of states; the temporal logic is used for expressing machine properties. However, the approach of Manna and Pnueli is based on first order logic. They propose two types of verification techniques [27] – a deductive approach, which provides a set of rules that reduce the task of proving a timing property to checking the validity of first-order formulas, and an algorithmic approach, which presents algorithms for automatic verification of timing properties. By contrast, in the DOVE system state-machine definitions are encoded as a set of axioms in the temporal logic. The sequent calculus method used is based on a deductive system for formal proof, and the deductive system is implemented in Isabelle HOL [33]. The temporal properties of a state machine can be verified by using the powerful HOL proof capabilities such as backward reasoning with tactics and tacticals.

Rather more different is the logic TLA, where just one language is used as opposed to the two in DOVE. In the framework based on TLA, systems and their properties are represented in the same logic. So, the assertion that a system design meets its specification, and the assertion that one system implements another, are both expressed by logical implication. Therefore, it is possible to formulate critical properties in a straightforward way. TLA also minimizes temporal reasoning, relying on ordinary, nontemporal reasoning whenever possible. TLA is suitable for specifying and reasoning about concurrent algorithms.

There are also other approaches to design verification that are based on using a single language for both design and requirements specification; two examples are Statecharts [16] and RSML [19]. However, using such a language makes it difficult to formulate the critical properties of a system in a sufficiently abstract way.

There also exists several impressive tools with aims similar to those of DOVE, such as KIV [39], Cogito [41] and PVS [37]. These are wider ranging in their application: they support reasoning in richer specification and design languages; and they can be used for a larger portion of the development phase. All of these tools support proof visualisation (for example, proof tree viewing), but none of them incorporate diagrammatic control of proofs in the application-domain level as the DOVE system does.

There are numerous existing examples of diagrammatic reasoning, but they differ from the examples of DOVE in several ways. Firstly, most of the existing mechanised support for formal reasoning using diagrams are developed for mathematical domains, rather than as notations for systems description such as state machines. Examples related to system designs do exist, such as [13], but mechanised support for reasoning in these frameworks is not well developed. Secondly, the popular approach in diagrammatic reasoning is to treat diagrams as first-class logical objects. This complicates the compilation of theories into an existing powerful proof assistant, as a theory of diagrams must then be formulated, in addition to the underlying semantic theory for the objects that are represented in diagrams.

In terms of fully-automated formal verification, the most successful approach has been that of model-checking [28, 35]. These techniques are being actively developed. However, such an approach is limited to finite state machines, or those that can be transformed to finite machines for verification purpose. The state-machine language described in this paper goes far beyond finite state machines. The typing language is very rich (that of Isabelle's HOL) and contains higher-order variables.

7.2 Outlook

The DOVE system uses Isabelle [34] to provide the basic logical and bookkeeping features for reasoning about state machine designs. The Isabelle system is a well-developed theorem-proving system with a large user base, and this application provides further evidence of its usefulness for software/system development purposes. The modelling of state machines and temporal properties in the DOVE tool also represents a significant application of Isabelle. Although Isabelle is natural deduction based, it has such great flexibility that there was little problem in constructing the sequent calculus over HOL expressions. The richness of the ML language allowed us to formulate easily the complicated tactics required, and to calculate values required for the graphical displays.

Extensions and improvements to the theoretical framework are being considered. Some of the future aims are listed here.

- A more expressive property language could be incorporated, as well as annotation of states with proved results which could then be used in later proofs.
- It should be possible to take into account structural features of state-machine graphs that users employ in reasoning but which are not currently supported by DOVE. The most notable example of this is the grouping of states and transitions into sub state-machines to provide a hierarchical structure. Currently a user can state and prove results about a disjunction of a number of states, but this can be quite awkward. Such a feature would be a useful extension to the tool, especially for reasoning about larger state machines. Thus we will also be incorporating hierarchies and abstraction, to facilitate understanding and reasoning about state machines.
- Further enhancements to the XIsabelle prover interface are also planned – and there has already been some progress on this front. The authors expect that great improvement in the usability of the DOVE tool can be achieved by addressing some basic problems with general proof management in XIsabelle, and Isabelle's subgoal package itself.
- The investigation of concurrent state machine designs has begun. This involves providing new state machine definitions that are suitable for expressing concurrency, and methodologies for modelling concurrency of such systems. We will also need to develop the appropriate proof system for verifying properties of concurrent state machines.
- It is planned to support "top-down" reasoning by allowing the user to prove refinement steps of a formal specification.
- A dataflow description of the high-level design could be incorporated to capture further informal understanding at this level.
- Probabilistic specifications, and stochastic dynamics for the state machines, are other extensions being investigated.

When completed, these extensions will bring the DOVE tool to the forefront of formal design.

References

1. Australian Department of Defence. *Def(Aust) Standard 5679: The procurement of Computer Based Safety Critical Systems*, 1998.

2. DOVE User Guide. Technical report, Trusted Computer Section, Information Technology Division, Defence Science and Technology Organisation, PO Box 1500, Salisbury, SA 5107, Australia, 1998.

3. B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

4. M. Baudinet, J. Chomicki, and P. Wolper. Temporal deductive databases. In A. Tansel *et al.*, editor, *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings Publishing Company, 1993.

5. C. Caleiro, G. Saake, and A. Sernadas. Deriving liveness goals from temporal logic specifications. *Journal of Symbolic Computation*, 22:521–553, 1996.

6. A. Cant, K. A. Eastaughffe, and M. A. Ozols. A tool for practical reasoning about state machine designs. In *Proceedings of 1996 Australian Software Engineering Conference*, pages 16–26. IEEE Computer Society Press, 1996.

7. E. Clarke, O. Grumberg, and R. Kurshan. A synthesis of two approaches for verifying finite state concurrent systems. *J. Logic and Computation*, 2:605–618, 1992.

8. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. Prog. Lang. Syst.*, 8:244–263, 1986.

9. E. W. Dijkstra. Formal verification of parallel programs. *Comm. ACM*, 19(7):371–384, 1976.

10. K. A. Eastaughffe, M. A. Ozols, and A. Cant. Proof tactics for a theory of state machines in a graphical environment. In *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, Lecture Notes in Artificial Intellegince, pages 366–379. Springer-Verlag, 1997.

11. European Communities – Commission. *ITSEC: Information Technology Security Evaluation Criteria*, 1991.

12. J. Fiadeiro and T. Maibaum. Temporal theories as modularization units for concurrent systems specification. *Formal Aspects of Computing*, 4:239–272, 1992.

13. K. Fisler. Exploiting the potential of diagrams in guiding hardware reasoning. In G. Allwein and J. Barwise, editors, *Logical Reasoning with Diagrams.* Oxford University Press, 1996.

14. D. M. Gabbay and P. McBrien. Temporal logic & historical databases. In *Proceedings of the 17th Very Large Data Bases Conference*, pages 423–430, Morgan Kauffman, 1991.

15. J.-R. Gagne and J. Plaice. A non-standard temporal deductive database system. *Journal of Symbolic Computation*, 22:649–664, 1996.

16. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Poloti, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Satetmate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–413, 1990.

17. L. Lamport. A simple approach to specifying concurrent systems. *Communications of ACM*, 32(1):32–45, 1989.

18. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

19. N. G. Leveson, M. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

20. D. Libes. *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs.* O'Reilly & Associate, Inc., 1995.

21. C. Liu and M. A. Orgun. A constraint mechanism for knowledge specification of simulation systems based on temporal logic. In *Advanced Topics in Artificial Intelligence*, Lecture Notes in Artificial Intelligence, Vol. 1342, pages 485–495. Springer-Verlag, 1997.

22. C. Liu and M. A. Orgun. Verification of reactive systems using temporal logic with clocks. *Theoretical Computer Science*, 220:377–408, 1999.

23. Z. Manna, A. Anuchitanukul, N. Bjorner, A. Browne, E. Change, M. Colón, L de Alfaro, H. Deverajan, H. Cipma, and T. Uribe. STeP: the Stanford Temporal Prover. Technical report, Department of Computer Science, Stanford University, June 1994.

24. Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In Boyer and Moore, editors, *Correctness Problem in Computer Science*, pages 215–273. Academic Press, 1981.

25. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag, 1992.

26. Z. Manna and A. Pnueli. A temporal proof methodology for reactive systems. In *Program Design Calculi – NATO ASI Series, Series F: Computer and System Science*. Springer-Verlag, 1993.

27. Z. Manna and A. Pnueli, editors. *Temporal Verification of Reactive Systems.* Springer-Verlag, 1995.

28. K. L. McMillan, editor. *Symbolic Model Checking.* Kluwer Academic, Boston, MA, 1993.

29. T. Nipkow. *The Isabelle/Isar Reference Manual.* `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`, 2000.

30. M. A. Orgun. A recursive temporal algebra and temporal completeness. In Luca Chittaro *et al.*, editor, *Proceedings of the Third International Workshop on Temporal Representation and Reasoning (TIME'96)*, pages 96–103. IEEE Computer Society Press, 1996.

31. J. K. Ousterhout, editor. *Tcl and Tk Toolkit.* Addison-Wesley, 1994.

32. M. A. Ozols, A. Cant, and K. A. Eastaughffe. XIsabelle: A system description. In *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, Lecture Notes in Artificial Intellegince, pages 400–403. Springer-Verlag, 1997.

33. L. C. Paulson. *ML for Working Programmer.* Cambridge University Press, 1991.

34. L. C. Paulson and T. Nipkow. In *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

35. A. W. Roscoe. Model checking csp. In *A Classical Mind, Essays in Honour of C. A. R. Hoare.* Prentice-Hall, 1994.

36. F. Sadri. Three approaches to temporal reasoning. In A. Galton, editor, *Temporal Logics and Their Applications*, pages 121–168. Academic Press, 1987.

37. N. Shankar. PVS: Combining specification, proof checking, and model checking. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 257–264. Springer-Verlag, 1996.

38. Y. Shoham, editor. *Reasoning About Change.* MIT Press, 1988.

39. K. Stenzel, W. Reif, and G. Schellhorn. Proving system correctness with KIV 3.0. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, pages 69–72, Townsville, Australia, July 1997. Springer-Verlag.

40. L. C. Paulson T. Nipkow and M. Wenzel. *Isabelle's Logics: HOL.* `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`, 2000.

41. A. Tuzhilin. SimTL: a simulation language based on temporal logic. *TRANSACTIONS of The Society of Computer Simulation*, 9(2):86–99, 1992.

42. UK Ministry of Defence. *Defence Standard 00-55: The procurement of Safety Critical Software in Defence Equipment*, 1995.

43. UK Ministry of Defence. *Defence Standard 00-56: The procurement of Safety Critical Software in Defence Equipment*, 1995.

44. M. Utting, R. Nickson, and O. Traynor. Cogito ergo sum – providing structured theorem prover support for specification formalisms. In *Proceedings of the 19st Australasian Computer Science Conference, ACSC'98*, pages 149–158, Melbourne, Australia, February 1998.

Appendix A Syntax for the state machine theory produced by DOVE

The language used in constructing a DOVE state machine is an extension of the Isabelle/Isar language. The reader unfamiliar with this syntax should consult [29]. Here we tailor our discussion to highlight the differences with and extensions of the basic Isar commands, although we will also include a description of the Isar commands found in the example of Subsection 3.1 for completeness. In the following:

- *id* is an identifier; i.e., a string of alphanumeric characters and underscore which must begin with a letter;
- *type* stands for any Isabelle type available in the state machine theory;
- *expr* stands for any mathematical expression (Isabelle term) in the variables and constants available in the state machine theory; and,

Moreover, we distinguish terminals of the grammar by enclosing them in single quote marks. In particular, we use the unusual device that ‘’ indicates a linebreak (or simply an empty line).

A.1 Extensions to the Isar command set

A.1.1 startmachine

initializes the theory, providing default values for the ML constructs which are produced.

A.1.2 states *statelist*

declares the state names of the state machine in the following syntax.

$$\begin{array}{lcl} \textit{statelist} & \longrightarrow & \textit{id} \\ & | & \textit{statelist}\ `|\text{'}\ \textit{id} \end{array}$$

The state identifiers are automatically produced from the user-drawn state-machine diagram. Of course, there are a finite number of state names. The other labels on the state-machine diagram are the transition names, which are declared implicitly in the **transdef** commands below.

A.1.3 inputs *inputdecl*

declares input variables with specified (value) type in the following syntax.

$$\begin{array}{lcl} \textit{inputdecl} & \longrightarrow & \textit{id}\ `\text{::}\text{'}\ \textit{type} \\ & | & \textit{inputdecl}\ `\text{'}\ \textit{id}\ `\text{::}\text{'}\ \textit{type} \end{array}$$

A.1.4 heaps *heapdecl*

declares heap variables with specified (value) type in the same syntax as for the **inputs** command.

A.1.5 initpred *expr*

declares the initial predicate of the state machine.

A.1.6 transdef *id tdefn*

declares a transition name, and the corresponding definition in the following syntax.

```
         tdefn  ⟶  let_expression
                   guard_part
                   action_part

let_expression  ⟶  ‘’
                |  LET assignmentlist
    guard_part  ⟶  ‘’
                |  GUARD expr
   action_part  ⟶  ‘’
                |  ACTION assignmentlist

assignmentlist  ⟶  id GETS expr‘;’
                |  assignmentlist id GETS expr‘;’

        ACTION  ⟶  ‘Act:’
          GETS  ⟶  ‘<--’
         GUARD  ⟶  ‘Guard:’
           LET  ⟶  ‘Let:’
```

A.1.7 graph *id edgelist*

declares the start state, and the list of edges in the state machine graph via the following syntax.

```
edgelist  ⟶  id ‘--’id‘-->’ id
          |  edgelist ‘|’ id ‘--’id‘-->’ id
```

A.1.8 endmachine

is called to ensure that all of the above declarations of the state machine are added into the state machine theory. The first step is to declare the configuration types, then the constants associated to input and heap variables, then the initialisation predicate, and finally all of the transitions. Transparent to the user, this command actually produces an Isabelle theory of the state machine in the abstract model described in Section 3.2.

A.2 Isar commands appearing in the TrafficLights theory

Beyond the command syntax introduced above, several standard Isar commands appear. Of course, this is just a small selection of the standard commands[5] – and the user is able to enter others beyond these. For completeness we first review their function, and comment on their use.

A.2.1 theory *thn* = DOVE+$thn_1 \ldots$

declares a new Isabelle/Isar theory *thn* built on a pre-existing theory, DOVE. Optional extra theory dependencies are added by listing them after DOVE, with separating +'s. Thus, the Isar syntax is maintained except that the "files" clause is not currently supported.

A.2.2 datatype $c_1 \mid c_2 \ldots$

is the simplest possible enumerated type definition. Isar allows much more general recursive definitions. DOVE also allows type synonyms, which would appear in the standard Isar **types** command. Noting that theories of type definitions can be included in the **theory** command, the allowed types in DOVE state machines are any type which can be produced in Isabelle; i.e., they are essentially arbitrary.

A.2.3 consts *cn* :: α

declares a constant *cn* to have any instance of the type scheme α. These are functions which the user introduces, any corresponding rules will appear in **axioms** commands as in standard Isar.

[5] For more details the user should consult [[illegible]].

Appendix B DOVE syntax for property specification

DOVE state-machine properties are written by the user in a concrete syntax which is summarized in the following syntax tree, up to the notational simplifications:

- "nontemporal" stands for any (boolean-valued) expression which can be written using the variables and constants (with their concrete types) defined in the user's theory, and from any included theories;
- *id* is an identifier, as used in Appendix A;
- *idts* stands for a list of identifiers with optional specification of type;
- where needed to disambiguate, the operator arguments are specified.

The user can employ the function application or lambda abstraction of the Isabelle HOL syntax as explained in [40]. The syntax tree is then straightforward to understand.

$$
\begin{array}{rcl}
\text{sequent} & \longrightarrow & \text{formulalist TURNSTILE formula} \\
 & | & \text{TURNSTILE formula} \\
 & & \\
\text{formulalist} & \longrightarrow & \text{formula} \\
 & | & \text{formulalist ‘,’ formula} \\
\text{formula} & \longrightarrow & \text{formula AND formula} \\
 & | & \text{formula OR formula} \\
 & | & \text{formula IMPLIES formula} \\
 & | & \text{formula CONGRUENCE formula} \\
 & | & \text{formula FROMTHENON formula} \\
 & | & \text{formula FROMTHENONS formula} \\
 & | & \text{MOSTRECENTLY formula formula} \\
 & | & \text{MOSTRECENTLYS formula formula} \\
 & | & \text{FORALL } \mathit{idts} \text{ . formula} \\
 & | & \text{EXISTS } \mathit{idts} \text{ . formula} \\
 & | & \text{ALWAYS formula} \\
 & | & \text{SOMETIME formula} \\
 & | & \text{formula} \\
 & | & \text{NOT formula} \\
 & | & \text{PREVIOUSLY formula} \\
 & | & \text{PREVIOUSLYS formula} \\
 & | & \text{INITIALLY formula} \\
 & | & \text{AT } \mathit{id} \\
 & | & \text{BY } \mathit{id} \\
 & | & \text{FIRST} \\
 & | & \text{TRUE} \\
 & | & \text{FALSE} \\
 & | & \text{nontemporal}
\end{array}
$$

```
              ALWAYS  ⟶  ‘[-]’
                       |  ‘Always’
                 AND  ⟶  ‘&’
                       |  ‘And’
                  AT  ⟶  ‘@-’
                       |  ‘At’
                  BY  ⟶  ‘>-’
                       |  ‘By’
          CONGRUENCE  ⟶  ‘<-->’
                       |  ‘EquivalentTo’
        EXISTS 1_ . 2_  ⟶  ‘? 1_ . 2_’
                       |  ‘Exists 1_ . 2_’
               FALSE  ⟶  ‘False’
               FIRST  ⟶  ‘first’
                       |  ‘First’
        FORALL 1_ . 2_  ⟶  ‘! 1_ . 2_’
                       |  ‘ForAll 1_ . 2_’
          FROMTHENON  ⟶  ‘~~>’
                       |  ‘FromThenOn’
         FROMTHENONS  ⟶  ‘~~>s’
                       |  ‘FromThenOnS’
             IMPLIES  ⟶  ‘-->’
                       |  ‘Implies’
           INITIALLY  ⟶  ‘init’
                       |  ‘Initially’
   MOSTRECENTLY 1_ 2_  ⟶  ‘<(1_) 2_’
                       |  ‘MostRecently 1_ 2_’
  MOSTRECENTLYS 1_ 2_  ⟶  ‘<S(1_) 2_’
                       |  ‘MostRecentlyS 1_ 2_’
                 NOT  ⟶  ‘~’
                       |  ‘Not’
                  OR  ⟶  ‘|’
                       |  ‘Or’
          PREVIOUSLY  ⟶  ‘(-)’
                       |  ‘Previously’
         PREVIOUSLYS  ⟶  ‘(S)’
                       |  ‘PreviouslyS’
            SOMETIME  ⟶  ‘<->’
                       |  ‘Sometime’
                TRUE  ⟶  ‘True’
          TURNSTILE  ⟶  ‘|-’
```

DISTRIBUTION LIST

State-Machine Modelling in the DOVE System

A. Cant, K. Eastaughffe[6], C. Liu,
B. Mahony, J. McCarthy and M. Ozols

	Number of Copies
DEFENCE ORGANISATION	
Task Sponsor	
Defence Signals Directorate, Carolyn Dyke	1
S&T Program	
Chief Defence Scientist	1
FAS Science Policy	
AS Science Corporate Management	
Director General Science Policy Development	
Counsellor, Defence Science, London	Doc Data Sht
Counsellor, Defence Science, Washington	Doc Data Sht
Scientific Adviser to MRDC, Thailand	Doc Data Sht
Scientific Adviser Joint	1
Navy Scientific Adviser	Doc Data Sht
Scientific Adviser, Army	Doc Data Sht
Air Force Scientific Adviser	1
Director Trials	1
Information Sciences Laboratory	
Chief of Information Networks Division	1
Research Leader of Information Assurance Branch	1
Head of Trusted Computer Systems Group	1
Task Manager	1
Author	1
DSTO Library and Archives	
Library Edinburgh	2
Australian Archives	1
Capability Systems Staff	
Director General Maritime Development	Doc Data Sht
Director General Land Development	Doc Data Sht
Director General Aerospace Development	Doc Data Sht

[6] Current address: Praxis Critical Systems Limited, 20 Manvers Street, Bath BA1 1PX, United Kingdom.

Knowledge Staff

Director General Command, Control, Communications and Computers	Doc Data Sht

Army

ABCA National Standardisation Officer, Land Warfare Development Sector, Puckapunyal	4
SO(Science), DJFHQ(L), Enoggera QLD	Doc Data Sht

Intelligence Program

DGSTA Defence Intelligence Organisation	1
Manager, Information Center, Defence Intelligence Organisation	1

Defence Libraries

Library Manager, DLS_Canberra	1
Library Manager, DLS_Sydney West	Doc Data Sht

UNIVERSITIES AND COLLEGES

Australian Defence Force Academy Library	1
Head of Aerospace and Mechanical Engineering, ADFA	1
Serials Section (M List), Deakin University Library, Geelong VIC	1
Hargrave Library, Monash University	Doc Data Sht
Librarian, Flinders University	1

OTHER ORGANISATIONS

National Library of Australia	1
NASA (Canberra)	1
AusInfo	1
State Library of South Australia	1

INTERNATIONAL DEFENCE INFORMATION CENTERS

US Defense Technical Information Center	2
UK Defence Research Information Center	2
Canada Defence Scientific Information Center	1
New Zealand Defence Information Center	1

ABSTRACTING AND INFORMATION ORGANISATIONS

Library, Chemical Abstracts Reference Service	1
Engineering Societies Library, US	1
Materials Information, Cambridge Science Abstracts, US	1

Documents Librarian, The Center for Research Libraries, US	1

INFORMATION EXCHANGE AGREEMENT PARTNERS

Acquisitions Unit, Science Reference and Information Service, UK	1
Library – Exchange Desk, National Institute of Standards and Technology, US	1

SPARES

	5
Total number of copies:	45

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. CAVEAT/PRIVACY MARKING

2. TITLE

State-Machine Modelling in the DOVE System

3. SECURITY CLASSIFICATION

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHORS

A. Cant, K. Eastaughffe[7], C. Liu,
B. Mahony, J. McCarthy and M. Ozols

5. CORPORATE AUTHOR

Information Sciences Laboratory
PO Box 1500
Edinburgh, South Australia, Australia 5111

6a. DSTO NUMBER

DSTO–RR–0255

6b. AR NUMBER

AR 012–783

6c. TYPE OF REPORT

Research Report

7. DOCUMENT DATE

February, 2003

8. FILE NUMBER

9. TASK NUMBER

JTW 02/106

10. SPONSOR

QR INFOSEC Branch, DSD

11. No OF PAGES

52

12. No OF REFS

44

13. URL OF ELECTRONIC VERSION

http://www.dsto.defence.gov.au/corporate/reports/DSTO–RR–0255.pdf

14. RELEASE AUTHORITY

Chief, Information Networks Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved For Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DEFTEST DESCRIPTORS

Modelling	Critical Systems
System Design	Verification

19. ABSTRACT

The DOVE tool supports high-level system modelling and design, and formal reasoning about critical properties. DOVE uses state-machine graphs to illustrate designs, thus building on a familiar and effective means of communicating system designs to a wide audience. DOVE employs a propositional temporal logic to express desirable behavioural properties of the design, and presents it in a sequent calculus syntax for ease of manipulation. A verification procedure which can handle temporal properties of DOVE state machines is included through high level tactics in a graphical proof tool interface. The DOVE program is committed to developing proof visualization techniques to complement the power of this proof scheme. This paper presents the theoretical structure underlying the DOVE tool.